SECURITY REVIEW REPORT FOR MANTLE

CONFIDENTIAL

CONFIDENTIAL

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

CONFIDENTIAL

LIMITATIONS ON DISCLOSURE AND USAGE OF THIS REPORT

This report has been developed by the company Hexens (the Service Provider) based on the Smart Contract Audit of Mantle (the Client). The document contains vulnerability information and remediation advice.

The information, presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Mantle.

If you are not the intended recipient of this document, remember that any disclosure, copying or dissemination of it is forbidden.

CONFIDENTIAL

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date
07.08.2023

Audit Completion Date
25.08.2023

---

hexens × MANTLE

METHODOLOGY

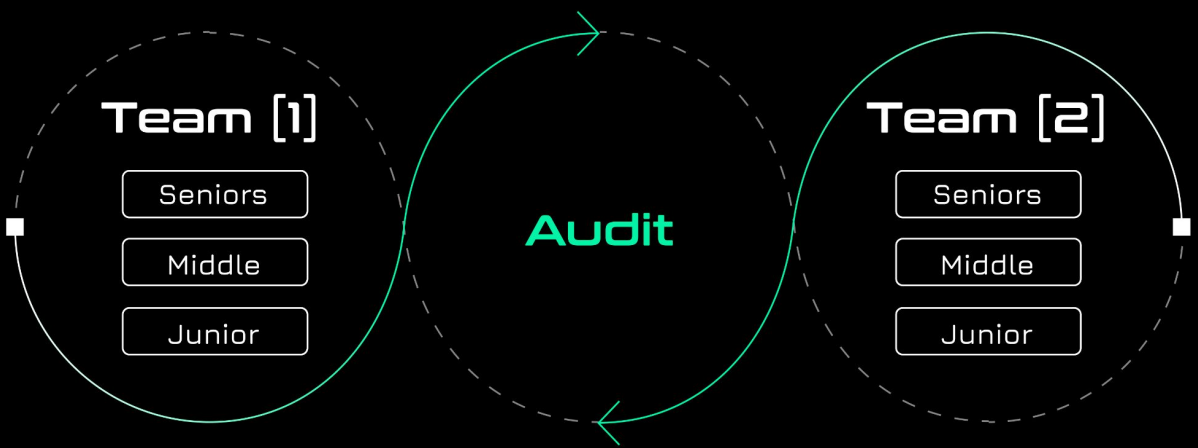
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

SCOPE

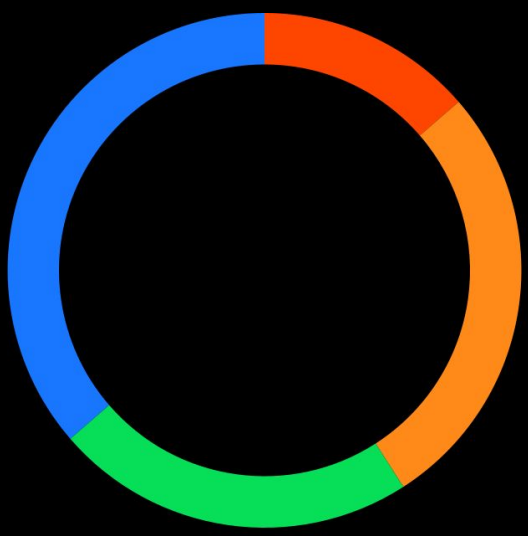
The analyzed resources were sent in an archive with the following SHA256 hash:
3c19735199673762f993a0ff6d9e04894a832ce4fc23d204999ad85669b4eb34

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	3
MEDIUM	6
LOW	5
INFORMATIONAL	8

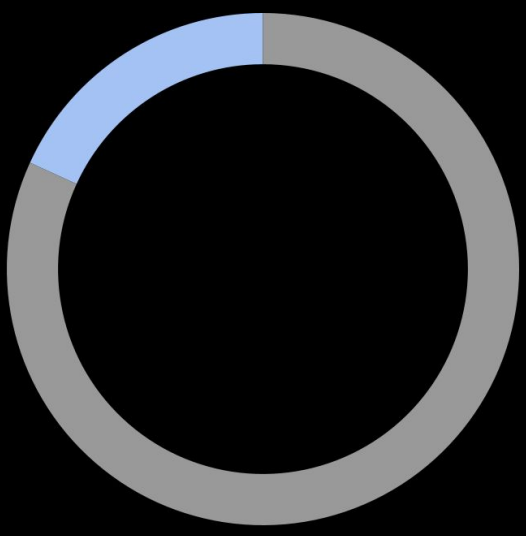
TOTAL: 22

SEVERITY



High Medium Low Informational

STATUS



New Disclosed

WEAKNESSES

This section contains the list of discovered weaknesses.

MAN1-8. MALICIOUS VALIDATOR CAN STEAL USER FUNDS BY FRONT-RUNNING WITHDRAWAL CREDENTIALS

SEVERITY: High

PATH: Staking.sol:initiateValidatorsWithDeposits:L396-457

REMEDIATION: this issue can be mitigated by either having validators pre-deposit 1 ETH with the protocol's withdrawal credentials and check this on-chain before adding it to Operator Registry

another mitigation (similar to Lido) is having the caller of deposit also supply the latest deposit state root from the DepositContract. This should then be checked before making the deposit. This ensures that there were no calls to the deposit contract between the signing and the actual depositing

STATUS: disclosed

DESCRIPTION:

By default in this protocol, the ones who have **INITIATOR_SERVICE_ROLE** and **STAKING_MANAGER_ROLE** manage the beacon chain validator creation process. The validator would sign the deposit data with the withdrawal credentials set to a protocol-controlled contract so that withdrawals would be received by the protocol (**withdrawalWallet_**).

In case the validator is malicious, they can monitor the mempool and front-run the **deposit** transaction of the beacon chain for its pubKey and deposit 1 ether for different withdrawal credentials. Because of the beacon chain implementation and ETH specs (link), in **ProcessDeposit**, if the **pubKey** is already registered, it increases its balance, not touching the **withdrawal_credentials**.

When the deposit reaches the consensus layer, the validator would be granted the 32 ETH of user funds and they can withdraw this to their own address.

This bug was also present in Lido and Frax:

Mitigations for deposit front-running vulnerability

Frontrunning by malicious validator · Issue #81 · code-423n4/2022-09-frax-findings

```
function initiateValidatorsWithDeposits(ValidatorParams[] calldata validators)
    external
    onlyRole(INITIATOR_SERVICE_ROLE)
{
    if (pauser.isInitiateValidatorsPaused()) {
        revert Paused();
    }
    if (validators.length == 0) {
        return;
    }

    // First loop is to check that all validators are valid according to our constraints and we record the
    // validators and how much we have deposited.
    uint256 amountDeposited = 0;
    for (uint256 i = 0; i < validators.length; ++i) {
        ValidatorParams calldata validator = validators[i];

        if (usedValidators[validator.pubkey]) {
            revert PreviouslyUsedValidator();
        }

        if (validator.depositAmount < minimumDepositAmount) {
            revert MinimumValidatorDepositNotSatisfied();
        }

        if (validator.depositAmount > maximumDepositAmount) {
            revert MaximumValidatorDepositExceeded();
        }

        if (validator.depositAmount > allocatedETHForDeposits) {
            revert NotEnoughDepositETH();
        }

        _requireProtocolWithdrawalAccount(validator.withdrawalCredentials);

        usedValidators[validator.pubkey] = true;
        amountDeposited += validator.depositAmount;

        emit ValidatorInitiated({
            id: keccak256(validator.pubkey),
            operatorID: validator.operatorID,
            pubkey: validator.pubkey,
            amountDeposited: validator.depositAmount
        });
    }
```

```
    allocatedETHForDeposits -= amountDeposited;
    totalDepositedInValidators += amountDeposited;
    numInitiatedValidators += validators.length;

    // Second loop is to send the deposits to the deposit contract. Keeps external calls to the deposit contract
    // separate from state changes.
    for (uint256 i = 0; i < validators.length; ++i) {
        ValidatorParams calldata validator = validators[i];
        depositContract.deposit{value: validator.depositAmount}({
            pubkey: validator.pubkey,
            withdrawal_credentials: validator.withdrawalCredentials,
            signature: validator.signature,
            deposit_data_root: validator.depositDataRoot
        });
    }
}
```

MAN1-11. STAKING DEPOSITS CAN BE BROKEN IMMEDIATELY AFTER DEPLOYMENT

SEVERITY: High

PATH: see description

REMEDIATION: the function ethToMntETH should also return ethAmount if the mntETH.totalSupply() == 0

STATUS: disclosed

DESCRIPTION:

The Oracle initialises the first oracle report with all 0 values by default in **initialize**. This allows the functionality such as **numRecords** and **recordAt** to work.

It also means that the ReturnAggregator report processing can be called from the beginning (which is a permissionless function). This function checks the balance of the ReturnReceiver and sends this to the Staking contract as rewards. This makes it possible for an attacker to send 1 Wei ETH to the ReturnReceiver, then call **processNextOracleRecords** on the ReturnAggregator.

As a result, the Staking will receive 1 Wei ETH through **receiveReturns** and this increases the **unallocatedETH** counter and subsequently the calculation in **totalControlled()** will now return 1 Wei ETH.

Now the share rate calculation in **ethToMntETH** passes the first check because **totalControlled()** is greater than zero and then it uses the total supply of 0 shares in the calculation, resulting in 0 shares for any deposit of any amount of ETH.

Because this function is used in **stake** to calculate the user's mintable shares, it becomes impossible to stake and it will block all future deposits. If users were to stake, they would just lose their ETH for 0 shares.

ReturnsAggregator.sol

```
function processNextOracleRecords(uint256 max) external {
    if (pauser.isProcessRecordsPaused()) {
        revert Paused();
    }

    uint256 num = oracle.numRecords() - numOracleRecordsProcessed;
    if (num > max) {
        num = max;
    }

    for (uint256 i = 0; i < num; i++) {
        _processNextOracleRecord();
    }
}
function _processNextOracleRecord() internal assertBalanceUnchanged {
    // As this is only called in `processNextOracleRecords`, which checks how many unprocessed records are
    // available, this should never fail.
    assert(numOracleRecordsProcessed < oracle.numRecords());

    OracleRecord memory record = oracle.recordAt(numOracleRecordsProcessed);
    // Updating this counter immediately even though the record was not fully processed yet to avoid any potential
    // reentrancy issues.
    numOracleRecordsProcessed++;

    // Calculate the total amount of returns that will be aggregated.
    uint256 elRewards = address(executionLayerReceiver).balance;
    uint256 clTotal = record.windowWithdrawnRewardAmount + record.windowWithdrawnPrincipalAmount;

    // Calculate protocol fees.
    uint256 totalRewards = elRewards + record.windowWithdrawnRewardAmount;
    uint256 fees = Math.mulDiv(feesBasisPoints, totalRewards, 10_000);

    // Aggregate returns in this contract
    address payable self = payable(address(this));
    if (elRewards > 0) {
        executionLayerReceiver.transfer(self, elRewards);
    }
    if (clTotal > 0) {
        consensusLayerReceiver.transfer(self, clTotal);
    }
```

```
    // Send protocol fees to the fee receiver wallet.
    Address.sendValue(feesReceiver, fees);

    // Forward the net returns to the staking contract.
    staking.receiveReturns{value: clTotal + elRewards - fees}();
}
```

Staking.sol

```
function receiveReturns() external payable onlyReturnsAggregator {
    unallocatedETH += msg.value;
}
function totalControlled() public view returns (uint256) {
    OracleRecord memory record = oracle.latestRecord();
    uint256 total = 0;
    total += unallocatedETH;
    total += allocatedETHForDeposits;
    /// The total ETH deposited to the beacon chain must be decreased by the deposits processed by the off-chain
    /// oracle since it will be accounted for in the currentTotalValidatorBalance from that point onwards.
    total += totalDepositedInValidators - record.cumulativeProcessedDepositAmount;
    total += record.currentTotalValidatorBalance;
    total += unstakeRequestsManager.balance();
    return total;
}
function ethToMntETH(uint256 ethAmount) public view returns (uint256) {
    // 1:1 if there is no controlled ETH.
    if (totalControlled() == 0) {
        return ethAmount;
    }
    return Math.mulDiv(
        ethAmount,
        mntETH.totalSupply() * uint256(_BASIS_POINTS_DENOMINATOR - exchangeAdjustmentRate),
        totalControlled() * uint256(_BASIS_POINTS_DENOMINATOR)
    );
}
```

MAN1-17. STEALING USER FUNDS DUE TO SKEWED SHARE RATE DURING WITHDRAWALS

SEVERITY: High

PATH: Staking.sol

REMEDIATION: the value of totalControlled() should be properly calculated during full validator withdrawal

this can be achieved by either having the total controlled ETH be calculated from the last fully processed oracle record only or to have the Oracle also call the ReturnsAggregator record processing function upon any oracle record finalisation

STATUS: disclosed

DESCRIPTION:

In the event of a full withdrawal of validator(s), the Oracle has a new record where **record.currentTotalValidatorBalanceis** reduced by the amount of **record.windowWithdrawnPrincipalAmount**.

The withdrawn ETH amount of **windowWithdrawnPrincipalAmount** should be transferred from the beacon chain to the **consensusLayerReceiver** (this is the contract **ReturnsReceiver.sol**).

Furthermore, someone should call the **processNextOracleRecords()** function of the **ReturnsAggregator** contract to apply changes from the oracle, where as a result withdrawn ether is added as **unallocatedEth** to the staking contract (L134, L153 of **ReturnsAggregator.sol**).

However, between the oracle record finalisation and the call to **processNextOraclerRecords**, there is a window of opportunity for an attacker to steal a share of the withdrawn funds.

After the record is accepted by the Oracle, the value **record.currentTotalValidatorBalance** inside the **totalControlled()** function is reduced (L527 of **Staking.sol**). Because the **totalControlled()** doesn't take into account **windowWithdrawnPrincipalAmount**, it temporarily returns a skewed value. Therefore, before the **processNextOracleRecords()** function is called, the **ethToMntETH()** reports a higher value than it should.

After oracle record finalisation with one or more withdrawn validators, the attacker can immediately deposit ETH into the staking contract at a discounted share rate and subsequently call **processNextOracleRecords** to restore the share rate, massively increasing the value of the minted share of the attacker.

The share of the stolen funds depends on the deposit of the attacker.

```
    function ethToMntETH(uint256 ethAmount) public view returns (uint256) {
        // 1:1 if there is no controlled ETH.
        if (totalControlled() == 0) {
            return ethAmount;
        }

        // deltaMntETH = (1 - exchangeAdjustmentRate) * (mntEthSupply / totalControlled) * ethAmount
        // This rounds down to zero in the case of `(1 - exchangeAdjustmentRate) * ethAmount * mntEthSupply <
        // totalControlled`.
        // While this scenario is theoretically possible, it can only be realised feasibly during the protocol's
        // bootstrap phase and if `totalControlled` and `mntEthSupply` can be changed independently of each other.
Since
        // the former is permissioned, and the latter is not permitted by the protocol, this cannot be exploited by an
        // attacker.
        return Math.mulDiv(
            ethAmount,
            mntETH.totalSupply() * uint256(_BASIS_POINTS_DENOMINATOR - exchangeAdjustmentRate),
            totalControlled() * uint256(_BASIS_POINTS_DENOMINATOR)
        );
    }
```

```
    function totalControlled() public view returns (uint256) {
        OracleRecord memory record = oracle.latestRecord();
        uint256 total = 0;
        total += unallocatedETH;
        total += allocatedETHForDeposits;
        /// The total ETH deposited to the beacon chain must be decreased by the deposits processed by the off-chain
        /// oracle since it will be accounted for in the currentTotalValidatorBalance from that point onwards.
        total += totalDepositedInValidators - record.cumulativeProcessedDepositAmount;
        total += record.currentTotalValidatorBalance;
        total += unstakeRequestsManager.balance();
        return total;
    }
```

MAN1-1. DEPOSIT SHARE RATE CAN BE MANIPULATED BY STAKING MANAGER

SEVERITY: Medium

PATH: Staking.sol:setExchangeAdjustmentRate:L599-610

REMEDIATION: to prevent such a situation, adjusting the validation check in the setExchangeAdjustmentRate function is recommended at least to disallow the exchangeAdjustmentRate_ from being a too large value (e.g. a maximum of 10% of _BASIS_POINTS_DENOMINATOR)

STATUS:

DESCRIPTION:

In the current implementation of the **setExchangeAdjustmentRate** function, there is a check for **exchangeAdjustmentRate_** value to be less or equal to the **_BASIS_POINTS_DENOMINATOR** (10,000). By setting **exchangeAdjustmentRate_** equal to **_BASIS_POINTS_DENOMINATOR**, the function **ethToMntETH** would always return a 0 value. This way users always will get a 0 amount of **mntETH** for staking.

The staking functionality also does not have any slippage protection and as such it leads to centralisation risk where the staking manager can deny people shares or steal ETH by front-running large deposits.

```
    function setExchangeAdjustmentRate(uint16 exchangeAdjustmentRate_) external
onlyRole(STAKING_MANAGER_ROLE) {
        if (exchangeAdjustmentRate_ > _BASIS_POINTS_DENOMINATOR) {
            revert InvalidConfiguration();
        }

        exchangeAdjustmentRate = exchangeAdjustmentRate_;
        emit ProtocolConfigChanged(
            this.setExchangeAdjustmentRate.selector,
            "setExchangeAdjustmentRate(uint16)",
            abi.encode(exchangeAdjustmentRate_)
        );
    }
```

```
    function ethToMntETH(uint256 ethAmount) public view returns (uint256) {
        // 1:1 if there is no controlled ETH.
        if (totalControlled() == 0) {
            return ethAmount;
        }

        // deltaMntETH = (1 - exchangeAdjustmentRate) * (mntEthSupply / totalControlled) * ethAmount
        // This rounds down to zero in the case of `(1 - exchangeAdjustmentRate) * ethAmount * mntEthSupply <
        // totalControlled`.
        // While this scenario is theoretically possible, it can only be realised feasibly during the protocol's
        // bootstrap phase and if `totalControlled` and `mntEthSupply` can be changed independently of each other.
Since
        // the former is permissioned, and the latter is not permitted by the protocol, this cannot be exploited by an
        // attacker.
        return Math.mulDiv(
            ethAmount,
            mntETH.totalSupply() * uint256(_BASIS_POINTS_DENOMINATOR - exchangeAdjustmentRate),
            totalControlled() * uint256(_BASIS_POINTS_DENOMINATOR)
        );
    }
```

MAN1-10. REWARDS ACCRUED BY UNSTAKE REQUESTS ARE NOT CORRECTLY DISTRIBUTED AMONG STAKERS

SEVERITY: Medium

PATH: UnstakeRequestsManager.sol:claim:L184-219

REMEDIATION: one way to mitigate this is to have the shares be burnt upon the creation of the withdrawal request, as well as the reserve the ETH equivalent in the totalControlledETH. The share rate would not change and any future rewards would be immediately distribute among the actual share holders during the withdrawal request

when the request is claimed, the reserved ETH would decrease with the original amount and the claimed ETH would increase with the finalised amount. Under normal circumstances the original amount and finalised amount are the same, so no rebase happens (this has already been done linearly over time)

only if the finalised amount is lower then the original amount (in case of slashing), would a negative rebase happen among the stakers at the time of claim. However, this is much less likely compared to the daily reward distribution

STATUS:

DESCRIPTION:

The UnstakeRequestsManager contract has a function **claim** where a user can claim the ETH of their finalised withdrawal request.

In this function, the finalised ETH is sent to the user and shares are burnt upon claim. The ETH is added to the **totalClaimed** variable, which has the effect that the total protocol-controlled ETH decreases.

The amount of ETH is according to the share rate from when the request was created and the shares that are being burned are worth more at the time of claim/burn (assuming normal conditions). This behaviour is correct in the sense that the requester should not get rewards from withdrawal requests

However, because the burn happens at the claim, it means that the rebase also happens at the time of claiming so the rewards are divided among stakers at that exact point in time, instead of linearly among stakers who were actually active during the unstake request.

This could potentially be exploited as an MEV strategy where a higher APR can be achieved by staking right before claims of large withdrawals.

```
function claim(uint256 requestID, address requester) external onlyStakingContract {
    UnstakeRequest memory request = _unstakeRequests[requestID];

    if (request.requester == address(0)) {
        revert AlreadyClaimed();
    }

    if (requester != request.requester) {
        revert NotRequester();
    }

    if (!_isFinalized(request)) {
        revert NotFinalized();
    }

    if (request.cumulativeETHRequested > allocatedETHForClaims) {
        revert NotEnoughFunds(request.cumulativeETHRequested, allocatedETHForClaims);
    }

    delete _unstakeRequests[requestID];
    totalClaimed += request.ethRequested;

    emit UnstakeRequestClaimed({
        id: requestID,
        requester: requester,
        mntEthLocked: request.mntEthLocked,
        ethRequested: request.ethRequested,
        cumulativeETHRequested: request.cumulativeETHRequested,
        blockNumber: request.blockNumber
    });

    // Claiming the request burns the locked MntETH tokens from this contract.
    mntETH.burn(request.mntEthLocked);

    Address.sendValue(payable(requester), request.ethRequested);
}
```

MAN1-13. MALICIOUS ORACLE REPORT CAN CAUSE DOS AND WRONG DISTRIBUTION OF FEES

SEVERITY: Medium

PATH: Oracle.sanityCheckUpdate:L331

REMEDIATION: the oracle consensus process has to be resistant against attacks where a malicious oracle report could be submitted. For example, the protocol has to ensure a large enough of quorum of individual and honest oracle nodes

we also recommend looking into EIP-4788 as discussed in issue 21

STATUS: disclosed

DESCRIPTION:

The Oracle contract accepts and finalises reports from Oracle nodes. It also has sanity checks for these oracle reports. We found that a malicious oracle report can make the sanity check on line 331 revert for any subsequent reporter.

This is possible by submitting a **newRecord** where **newRecord.cumulativeNumValidatorsFullyWithdrawn** contains a higher value than it should. Consequently the first check in the **sanityCheckUpdate** function will revert on the next call when **newRecord.cumulativeNumValidatorsFullyWithdrawn** < **prevRecord.cumulativeNumValidatorsFullyWithdrawn** (L339) is called with what were supposed to be correct values.

Next in the execution flow, when checking **if (newNumValidators < prevNumValidators)** it is important to note that a malicious reporter could pass **newRecord.currentNumValidatorsWithBalance = 0** and **newRecord.cumulativeNumValidatorsFullyWithdrawn** with any value necessary to satisfy the check.

On the next check in line 375 the same can occur when checking **if (newDeposits < newValidators * minDepositPerValidator)** where the variable **newValidators** includes both **newRecord.currentNumValidatorsWithBalance + newRecord.cumulativeNumValidatorsFullyWithdrawn**

The **upperBound** and **lowerBoundchecks** between lines 400-421 can be easily passed by an attacker that chooses a value for **newRecord.windowWithdrawnPrincipalAmount** that satisfies the requirements.

In order to pass the last two checks related to **newGrossCLBalance** the attacker can pass in any value for **newRecord.windowWithdrawnRewardAmount**.

Now the **sanityCheckUpdate** has passed even though **cumulativeNumValidatorsFullyWithdrawn** is higher than it should be. Therefore, the next reporter who calls **receiveReport** with the right expected values will experience a revert due to the check in line 339:

newRecord.cumulativeNumValidatorsFullyWithdrawn < prevRecord.cumulativeNumValidatorsFullyWithdrawn

Depending on **newRecord.windowWithdrawnPrincipalAmount** and **newRecord.windowWithdrawnRewardAmount** values this attack could also lead to the wrong amount of being sent to **feesReceiver** (receiving less or none) and **staking** contract (receiving more) via the **processNextOracleRecords** function if **executionLayerReceiver** and **consensusLayerReceiver** contain enough balance for the transaction.

modifyExistingRecord could be called to rectify the malicious behaviour and undo the DOS, however the the wrong amount of **fees** and **clTotal + elRewards - fee** would already have been sent to the contracts.

Note: This scenario assumes **absoluteThreshold = 1**. In case **absoluteThreshold > 1** there would either have to be a coordinate attack between reporters or a hash collision would have to be manufactured for attack to be successful.

```
function sanityCheckUpdate(OracleRecord calldata newRecord) public view returns (string memory, uint256, uint256) {
    OracleRecord memory prevRecord = latestRecord();
    {
      //
      // Number of validators
      //
      // Checks the total number of validators that were withdrawn or added to the consensus layer
      // and verifies they did not decrease in the new oracle period.
      if (newRecord.cumulativeNumValidatorsFullyWithdrawn < prevRecord.cumulativeNumValidatorsFullyWithdrawn) {
        return (
          "Number of fully withdrawn validators decreased",
          newRecord.cumulativeNumValidatorsFullyWithdrawn,
          prevRecord.cumulativeNumValidatorsFullyWithdrawn
        );
      }
      {
        uint256 prevNumValidators =
          prevRecord.currentNumValidatorsWithBalance + prevRecord.cumulativeNumValidatorsFullyWithdrawn;
        uint256 newNumValidators =
          newRecord.currentNumValidatorsWithBalance + newRecord.cumulativeNumValidatorsFullyWithdrawn;

        if (newNumValidators < prevNumValidators) {
          return ("Total number of validators decreased", newNumValidators, prevNumValidators);
        }
      }
    }

    {
      //
      // Deposits
      //
      // Checks that the total amount of deposits processed by the oracle did not decrease in the new oracle
      // period. It also checks that the amount of newly deposited ETH is possible given how many validators
      // we have included in the new period.
      if (newRecord.cumulativeProcessedDepositAmount < prevRecord.cumulativeProcessedDepositAmount) {
        return (
          "Processed deposit amount decreased",
          newRecord.cumulativeProcessedDepositAmount,
          prevRecord.cumulativeProcessedDepositAmount
        );
      }

      uint256 newDeposits =
        (newRecord.cumulativeProcessedDepositAmount - prevRecord.cumulativeProcessedDepositAmount);
      uint256 newValidators = (
        newRecord.currentNumValidatorsWithBalance + newRecord.cumulativeNumValidatorsFullyWithdrawn
          - prevRecord.currentNumValidatorsWithBalance - prevRecord.cumulativeNumValidatorsFullyWithdrawn
      );
```

```
    if (newDeposits < newValidators * minDepositPerValidator) {
      return (
        "New deposits below min deposit per validator", newDeposits, newValidators * minDepositPerValidator
      );
    }

    if (newDeposits > newValidators * maxDepositPerValidator) {
      return (
        "New deposits above max deposit per validator", newDeposits, newValidators * maxDepositPerValidator
      );
    }
  }

  {
    //
    // Withdrawn Principals
    //
    // Checks the bounds on withdrawn principal amount given by the number of validators that did a full
    // withdrawal. This can help in the case of a misclassification of withdrawals with rewards.
    {
      uint256 upperBound = maxConsensusLayerBalancePerValidator
        * (newRecord.cumulativeNumValidatorsFullyWithdrawn - prevRecord.cumulativeNumValidatorsFullyWithdrawn);
      if (newRecord.windowWithdrawnPrincipalAmount > upperBound) {
        return (
          "Withdrawn principal amount above bounds per validator",
          newRecord.windowWithdrawnPrincipalAmount,
          upperBound
        );
      }
    }
    {
      uint256 lowerBound = minConsensusLayerBalancePerValidator
        * (newRecord.cumulativeNumValidatorsFullyWithdrawn - prevRecord.cumulativeNumValidatorsFullyWithdrawn);
      if (newRecord.windowWithdrawnPrincipalAmount < lowerBound) {
        return (
          "Withdrawn principal amount below bounds per validator",
          newRecord.windowWithdrawnPrincipalAmount,
          lowerBound
        );
      }
    }
  }
```

```
{
    //
    // Consensus layer balance change from the previous period.
    //
    // Checks that the change in the consensus layer balance is within the bounds given by the maximum loss and
    // minimum gain parameters. For example, a major slashing event will cause an out of bounds loss in the
    // consensus layer.

    // The baselineGrossCLBalance represents the expected growth of our validators balance in the new period
    // given no slashings, no rewards, etc. It's used as the baseline in our upper (growth) and lower (loss)
    // bounds calculations.
    uint256 baselineGrossCLBalance = prevRecord.currentTotalValidatorBalance
      + (newRecord.cumulativeProcessedDepositAmount - prevRecord.cumulativeProcessedDepositAmount);

    // The newGrossCLBalance is the actual amount of ETH we have recorded in the consensus layer for the new
    // record period.
    uint256 newGrossCLBalance = newRecord.currentTotalValidatorBalance
      + newRecord.windowWithdrawnPrincipalAmount + newRecord.windowWithdrawnRewardAmount;

    {
      // Relative lower bound on the net decrease of ETH on the consensus layer.
      // Depending on the parameters the loss term might completely dominate over the minGain one.
      //
      // Using a minConsensusLayerGainPerBlockPPT greater than 0, the lower bound becomes an upward slope.
      // Setting minConsensusLayerGainPerBlockPPT, the lower bound becomes a constant.
      uint256 lowerBound = baselineGrossCLBalance
        - Math.mulDiv(maxConsensusLayerLossPPM, baselineGrossCLBalance, _PPM_DENOMINATOR)
        + Math.mulDiv(
          minConsensusLayerGainPerBlockPPT * (newRecord.updateEndBlock - newRecord.updateStartBlock),
          baselineGrossCLBalance,
          _PPT_DENOMINATOR
        );

      if (newGrossCLBalance < lowerBound) {
        return ("Consensus layer change below min gain or max loss", newGrossCLBalance, lowerBound);
      }
    }
    {
      // Upper bound on the rewards generated by validators scaled linearly with time and number of active
      // validators.
      uint256 upperBound = baselineGrossCLBalance
        + Math.mulDiv(
          maxConsensusLayerGainPerBlockPPT * (newRecord.updateEndBlock - newRecord.updateStartBlock),
          baselineGrossCLBalance,
          _PPT_DENOMINATOR
        );
```

```
            if (newGrossCLBalance > upperBound) {
                return ("Consensus layer change above max gain", newGrossCLBalance, upperBound);
            }
        }
    }

    return ("", 0, 0);
}
```

MAN1-18. MALICIOUS ORACLE REPORT CAN MANIPULATE THE SHARE RATE FOR PROFIT

SEVERITY: Medium

PATH: Oracle.sol

REMEDIATION: the oracle consensus process has to be resistant against attacks where a malicious oracle report could be submitted. For example, the protocol has to ensure a large enough of quorum of individual and honest oracle nodes

we also recommend looking into EIP-4788 as discussed in issue 21

STATUS:

DESCRIPTION:

Malicious Oracle reporter can submit in **receiveRecord()** that all validators have been fully withdrawn. By setting **newRecord.currentTotalValidatorBalance** to **0** and **newRecord.windowWithdrawnPrincipalAmount** to the value that the sum of **newRecord.currentTotalValidatorBalance + newRecord.windowWithdrawnPrincipalAmount** looks plausible and the value of the **newGrossCLBalance** lies between **lowerBound** and **upperBound** (L439-L473 of **Oracle.sol**).

```
uint256 newGrossCLBalance = newRecord.currentTotalValidatorBalance
        + newRecord.windowWithdrawnPrincipalAmount + newRecord.windowWithdrawnRewardAmount;
```

And also **newRecord.windowWithdrawnPrincipalAmount** should be inside a specific range (L399-L420 of **Oracle.sol**).

Finally all checks for the Oracle record are passed and the **newRecord** is added.

As a result, **processNextOracleRecords()** of **ReturnsAggregator.sol** will revert (L145-L147 of **ReturnsAggregator.sol**) and **totalControlled()** of **Staking.sol** plummets, because **record.currentTotalValidatorBalance** is **0** now (L527 of **Staking.sol)** and **unallocatedETH** is low (L522 of **Staking.sol**).

Consequentially, the malicious Oracle reporter can make a go out of high ethToMntETH rate by minting enormous amount of mntETH tokens.

```
    function totalControlled() public view returns (uint256) {
        OracleRecord memory record = oracle.latestRecord();
        uint256 total = 0;
        total += unallocatedETH;
        total += allocatedETHForDeposits;
        /// The total ETH deposited to the beacon chain must be decreased by the deposits processed by
the off-chain
        /// oracle since it will be accounted for in the currentTotalValidatorBalance from that point onwards.
        total += totalDepositedInValidators - record.cumulativeProcessedDepositAmount;
        total += record.currentTotalValidatorBalance;
        total += unstakeRequestsManager.balance();
        return total;
    }
```

```
function sanityCheckUpdate(OracleRecord calldata newRecord) public view returns (string memory, uint256, uint256) {
    OracleRecord memory prevRecord = latestRecord();
    {
        //
        // Number of validators
        //
        // Checks the total number of validators that were withdrawn or added to the consensus layer
        // and verifies they did not decrease in the new oracle period.
        if (newRecord.cumulativeNumValidatorsFullyWithdrawn < prevRecord.cumulativeNumValidatorsFullyWithdrawn) {
            return (
                "Number of fully withdrawn validators decreased",
                newRecord.cumulativeNumValidatorsFullyWithdrawn,
                prevRecord.cumulativeNumValidatorsFullyWithdrawn
            );
        }
        {
            uint256 prevNumValidators =
                prevRecord.currentNumValidatorsWithBalance + prevRecord.cumulativeNumValidatorsFullyWithdrawn;
            uint256 newNumValidators =
                newRecord.currentNumValidatorsWithBalance + newRecord.cumulativeNumValidatorsFullyWithdrawn;

            if (newNumValidators < prevNumValidators) {
                return ("Total number of validators decreased", newNumValidators, prevNumValidators);
            }
        }
    }

    {
        //
        // Deposits
        //
        // Checks that the total amount of deposits processed by the oracle did not decrease in the new oracle
        // period. It also checks that the amount of newly deposited ETH is possible given how many validators
        // we have included in the new period.
        if (newRecord.cumulativeProcessedDepositAmount < prevRecord.cumulativeProcessedDepositAmount) {
            return (
                "Processed deposit amount decreased",
                newRecord.cumulativeProcessedDepositAmount,
                prevRecord.cumulativeProcessedDepositAmount
            );
        }
```

```
        uint256 newDeposits =
            (newRecord.cumulativeProcessedDepositAmount - prevRecord.cumulativeProcessedDepositAmount);
        uint256 newValidators = (
            newRecord.currentNumValidatorsWithBalance + newRecord.cumulativeNumValidatorsFullyWithdrawn
                - prevRecord.currentNumValidatorsWithBalance - prevRecord.cumulativeNumValidatorsFullyWithdrawn
        );

        if (newDeposits < newValidators * minDepositPerValidator) {
            return (
                "New deposits below min deposit per validator", newDeposits, newValidators * minDepositPerValidator
            );
        }

        if (newDeposits > newValidators * maxDepositPerValidator) {
            return (
                "New deposits above max deposit per validator", newDeposits, newValidators * maxDepositPerValidator
            );
        }
    }

    {
        //
        // Withdrawn Principals
        //
        // Checks the bounds on withdrawn principal amount given by the number of validators that did a full
        // withdrawal. This can help in the case of a misclassification of withdrawals with rewards.
        {
            uint256 upperBound = maxConsensusLayerBalancePerValidator
                * (newRecord.cumulativeNumValidatorsFullyWithdrawn - prevRecord.cumulativeNumValidatorsFullyWithdrawn);
            if (newRecord.windowWithdrawnPrincipalAmount > upperBound) {
                return (
                    "Withdrawn principal amount above bounds per validator",
                    newRecord.windowWithdrawnPrincipalAmount,
                    upperBound
                );
            }
        }
```

```
    {
        uint256 lowerBound = minConsensusLayerBalancePerValidator
            * (newRecord.cumulativeNumValidatorsFullyWithdrawn - prevRecord.cumulativeNumValidatorsFullyWithdrawn);
        if (newRecord.windowWithdrawnPrincipalAmount < lowerBound) {
            return (
                "Withdrawn principal amount below bounds per validator",
                newRecord.windowWithdrawnPrincipalAmount,
                lowerBound
            );
        }
    }
}

{
    //
    // Consensus layer balance change from the previous period.
    //
    // Checks that the change in the consensus layer balance is within the bounds given by the maximum loss and
    // minimum gain parameters. For example, a major slashing event will cause an out of bounds loss in the
    // consensus layer.

    // The baselineGrossCLBalance represents the expected growth of our validators balance in the new period
    // given no slashings, no rewards, etc. It's used as the baseline in our upper (growth) and lower (loss)
    // bounds calculations.
    uint256 baselineGrossCLBalance = prevRecord.currentTotalValidatorBalance
        + (newRecord.cumulativeProcessedDepositAmount - prevRecord.cumulativeProcessedDepositAmount);

    // The newGrossCLBalance is the actual amount of ETH we have recorded in the consensus layer for the new
    // record period.
    uint256 newGrossCLBalance = newRecord.currentTotalValidatorBalance
        + newRecord.windowWithdrawnPrincipalAmount + newRecord.windowWithdrawnRewardAmount;

    {
        // Relative lower bound on the net decrease of ETH on the consensus layer.
        // Depending on the parameters the loss term might completely dominate over the minGain one.
        //
        // Using a minConsensusLayerGainPerBlockPPT greater than 0, the lower bound becomes an upward slope.
        // Setting minConsensusLayerGainPerBlockPPT, the lower bound becomes a constant.
        uint256 lowerBound = baselineGrossCLBalance
            - Math.mulDiv(maxConsensusLayerLossPPM, baselineGrossCLBalance, _PPM_DENOMINATOR)
            + Math.mulDiv(
                minConsensusLayerGainPerBlockPPT * (newRecord.updateEndBlock - newRecord.updateStartBlock),
                baselineGrossCLBalance,
                _PPT_DENOMINATOR
            );
```

```
            if (newGrossCLBalance < lowerBound) {
                return ("Consensus layer change below min gain or max loss", newGrossCLBalance, lowerBound);
            }
        }
        {
            // Upper bound on the rewards generated by validators scaled linearly with time and number of active
            // validators.
            uint256 upperBound = baselineGrossCLBalance
                + Math.mulDiv(
                    maxConsensusLayerGainPerBlockPPT * (newRecord.updateEndBlock - newRecord.updateStartBlock),
                    baselineGrossCLBalance,
                    _PPT_DENOMINATOR
                );

            if (newGrossCLBalance > upperBound) {
                return ("Consensus layer change above max gain", newGrossCLBalance, upperBound);
            }
        }
    }

    return ("", 0, 0);
}
```

MAN1-19. FEE RECEIVER IN RETURNSAGGREGATOR CAN STEAL USER FUNDS

SEVERITY: Medium

PATH: ReturnsAggregator.sol:_processNextOracleRecord:L122-154

REMEDIATION: address.sendValue should be called at the end of the _processNextOracleRecord function

STATUS:

DESCRIPTION:

In the function where the ReturnsAggregator processes the next oracle record, some amount of ETH is sent to the fee receiver by calling **sendValue()** on line 150. This happens before before sending ETH to the staking contract, thus violating the Check Effect Interactions (CEI) pattern.

If one or more validators have withdrawn and this ETH has come back to the protocol, then when this fee is sent, the Staking contract is in a state where the total controlled ETH is decreased by the withdrawn amount, as described in issue 17.

A malicious fee receiver could still exploit this issue and take advantage of the skewed balance of the staking contract for profit.

```
  function _processNextOracleRecord() internal assertBalanceUnchanged {
      // As this is only called in `processNextOracleRecords`, which checks how many unprocessed
records are
      // available, this should never fail.
      assert(numOracleRecordsProcessed < oracle.numRecords());

      OracleRecord memory record = oracle.recordAt(numOracleRecordsProcessed);
      // Updating this counter immediately even though the record was not fully processed yet to avoid
any potential
      // reentrancy issues.
      numOracleRecordsProcessed++;

      // Calculate the total amount of returns that will be aggregated.
      uint256 elRewards = address(executionLayerReceiver).balance;
      uint256 clTotal = record.windowWithdrawnRewardAmount + record.windowWithdrawnPrincipalAmount;

      // Calculate protocol fees.
      uint256 totalRewards = elRewards + record.windowWithdrawnRewardAmount;
      uint256 fees = Math.mulDiv(feesBasisPoints, totalRewards, 10_000);

      // Aggregate returns in this contract
      address payable self = payable(address(this));
      if (elRewards > 0) {
          executionLayerReceiver.transfer(self, elRewards);
      }
      if (clTotal > 0) {
          consensusLayerReceiver.transfer(self, clTotal);
      }

      // Send protocol fees to the fee receiver wallet.
      Address.sendValue(feesReceiver, fees);

      // Forward the net returns to the staking contract.
      staking.receiveReturns{value: clTotal + elRewards - fees}();
  }
```

MAN1-23. NO SLIPPAGE CHECKS ON DEPOSIT AND WITHDRAW

SEVERITY: Medium

PATH: Staking.sol

REMEDIATION: we would recommend to either add a parameter to stake such as minShares that checks that the amount of minted shares is not lower than the minimum as defined by the user. Similarly unstakeRequest a parameter such as maxShares to check the maximum amount of burned shares

if no changes to these functions are desired, we would recommend to deploy a periphery contract in front of the staking contract that employs this slippage protection. This periphery contract should then be used by the front-end and other users

STATUS:

DESCRIPTION:

In the Staking contract, both the deposit function **stake** and withdraw function **unstakeRequest** have no slippage protection and could be vulnerable to front-running with MEV strategies that manipulate the share rate (such as issue 1 and 17), potentially causing a loss to the user.

```
function stake() external payable {
    if (pauser.isStakingPaused()) {
        revert Paused();
    }

    if (isStakingAllowlist) {
        _checkRole(STAKING_ALLOWLIST_ROLE);
    }

    if (msg.value < minimumStakeBound) {
        revert MinimumStakeBoundNotSatisfied();
    }

    uint256 mntETHMintAmount = ethToMntETH(msg.value);

    // Increment unallocated ETH after calculating the exchange rate to ensure
    // a consistent rate.
    unallocatedETH += msg.value;

    emit Staked(msg.sender, msg.value, mntETHMintAmount);
    mntETH.mint(msg.sender, mntETHMintAmount);
}

function unstakeRequest(uint128 mntETHAmount) external returns (uint256) {
    return _unstakeRequest(mntETHAmount);
}
```

MAN1-2. STAKING VALIDATOR DEPOSIT REDUNDANT CHECKS AND VARIABLES

SEVERITY: Low

PATH: Staking.sol:initiateValidatorsWithDeposits:L396-457

REMEDIATION: see description

STATUS:

DESCRIPTION:

In the Staking contract, the function to deposit ETH in to the DepositContract and create validators uses 2 storage variables **minimumDepositAmount** and **maximumDepositAmount** to check the specified **validator.depositAmount** bounds.

Both variables are currently set at 32 ETH and validators can only be activated once total deposits for the pubkey reaches 32 ETH.

Therefore, having a check for **validator.deposit** being less than **minimumDepositAmount** and bigger than **maximumDepositAmount** is redundant and could be refactored to one equality check of deposit size.

```
if (validator.depositAmount < minimumDepositAmount) {
    revert MinimumValidatorDepositNotSatisfied();
}

if (validator.depositAmount > maximumDepositAmount) {
    revert MaximumValidatorDepositExceeded();
}
```

```
minimumDepositAmount = 32 ether;
maximumDepositAmount = 32 ether;
```

Replace the checks with only one equality check that check the value is equal to 32 ETH.

```
if (validator.depositAmount != 32 ether) {
    revert WrongValidatorDepositAmount();
}
```

MAN1-6. CENTRALISATION RISK FROM PAUSEABLE UNSTAKE AND CLAIM FUNCTIONALITY

SEVERITY: Low

PATH: Staking.sol:_unstakeRequest, claimUnstakeRequest

REMEDIATION: consider removing any pausing functionality from the functions_unstakeRequest and claimUnstakeRequest in the Staking contract

STATUS:

DESCRIPTION:

When a user wants to unstake, this is initiated in the Staking contract and passed to the UnstakeRequestsManager. The functions **_unstakeRequest** and **claimUnstakeRequest** exist to facilitate this.

However, these functions can be paused through **isUnstakeRequestsAndClaimsPaused**. In some cases, this is not recommendation because the ability to freeze user funds can cause reputational damage to the protocol.

In addition, if the protocol is paused and **PAUSER_ROLE** keys are lost or compromised the funds would potentially remain frozen.

```
function setIsUnstakeRequestsAndClaimsPaused(bool isPaused) external onlyPauserUnpauserRole(isPaused) {
    _setIsUnstakeRequestsAndClaimsPaused(isPaused);
}
```

MAN1-12. FINALIZATIONBLOCKNUMBERDELTA SHOULD HAVE UPPER BOUND

SEVERITY: Low

PATH: Oracle.sol:setFinalizationBlockNumberDelta:L548

REMEDIATION: implement a check to ensure finalizationBlockNumberDelta is not a unreasonably high number which would cause all reports to revert

STATUS:

DESCRIPTION:

The function to set **finalizationBlockNumberDelta** should have a sensible upper limit so the function **receiveRecord** does not end up reverting on every call for an indefinite period of time due to lack of input sanitisation.

For example, accidentally typing one extra digit when calling **setFinalizationBlockNumberDelta** could be enough to DoS the protocol for some period of time.

```
function setFinalizationBlockNumberDelta(uint256 finalizationBlockNumberDelta_)
    external
    onlyRole(ORACLE_MANAGER_ROLE)
{
    if (finalizationBlockNumberDelta_ == 0) {
        revert InvalidConfiguration();
    }
    finalizationBlockNumberDelta = finalizationBlockNumberDelta_;
    emit ProtocolConfigChanged(
        this.setFinalizationBlockNumberDelta.selector,
        "setFinalizationBlockNumberDelta(uint256)",
        abi.encode(finalizationBlockNumberDelta_)
    );
}
```

```
    uint256 updateFinalizingBlock = newRecord.updateEndBlock + finalizationBlockNumberDelta;
    if (block.number < updateFinalizingBlock) {
        revert UpdateEndBlockNumberNotFinal(updateFinalizingBlock);
    }
```

MAN1-22. EXCHANGE ADJUSTMENT RATE HAS NO DEFAULT VALUE

SEVERITY: Low

PATH: Staking.sol

REMEDIATION: set a non zero default value for the exchangeAdjustmentRate

STATUS:

DESCRIPTION:

The state variable **exchangeAdjustmentRate** participates in the calculation of the **ethToMntETH** rate thus compensating the difference between the length of entry and exit queues for staking on the beacon chain. Additionally, it helps to alleviate griefing attacks against the protocol.

However, the default value for the **exchangeAdjustmentRate** is **0** and the setter **setExchangeAdjustmentRate()** should be explicitly called to change its default value. Moreover, deployment scripts don't call **setExchangeAdjustmentRate()**.

```
    function ethToMntETH(uint256 ethAmount) public view returns (uint256) {
        // 1:1 if there is no controlled ETH.
        if (totalControlled() == 0) {
            return ethAmount;
        }

        // deltaMntETH = (1 - exchangeAdjustmentRate) * (mntEthSupply / totalControlled) * ethAmount
        // This rounds down to zero in the case of `(1 - exchangeAdjustmentRate) * ethAmount * mntEthSupply <
        // totalControlled`.
        // While this scenario is theoretically possible, it can only be realised feasibly during the protocol's
        // bootstrap phase and if `totalControlled` and `mntEthSupply` can be changed independently of each other. Since
        // the former is permissioned, and the latter is not permitted by the protocol, this cannot be exploited by an
        // attacker.
        return Math.mulDiv(
            ethAmount,
            mntETH.totalSupply() * uint256(_BASIS_POINTS_DENOMINATOR - exchangeAdjustmentRate),
            totalControlled() * uint256(_BASIS_POINTS_DENOMINATOR)
        );
    }
```

MAN1-24. REDUNDANT VALUE CHECK IN VALIDATOR DEPOSIT

SEVERITY: Low

PATH: Staking.sol:initiateValidatorsWithDeposits:L396-457

REMEDIATION: see description

STATUS:

DESCRIPTION:

The function **initiateValidatorsWithDeposits** is used to deposit ETH to the DepositContract and create new validators. There is a loop over each entry in the **validators** parameter, where the **validator.depositAmount** is checked against the **allocatedETHForDeposits**.

However, this is incorrect, as after the loop the total sum of each **depositAmount** is subtracted from the **allocatedETHForDeposits**. The current check only checks if each individual **depositAmount** is smaller or equal to the total available ETH, while it should check the total sum after the loop.

As a result, there is a redundant check in each iteration.

```
function initiateValidatorsWithDeposits(ValidatorParams[] calldata validators)
    external
    onlyRole(INITIATOR_SERVICE_ROLE)
{
    if (pauser.isInitiateValidatorsPaused()) {
        revert Paused();
    }
    if (validators.length == 0) {
        return;
    }

    // First loop is to check that all validators are valid according to our constraints and we record the
    // validators and how much we have deposited.
    uint256 amountDeposited = 0;
    for (uint256 i = 0; i < validators.length; ++i) {
        ValidatorParams calldata validator = validators[i];

        if (usedValidators[validator.pubkey]) {
            revert PreviouslyUsedValidator();
        }

        if (validator.depositAmount < minimumDepositAmount) {
            revert MinimumValidatorDepositNotSatisfied();
        }

        if (validator.depositAmount > maximumDepositAmount) {
            revert MaximumValidatorDepositExceeded();
        }

        if (validator.depositAmount > allocatedETHForDeposits) {
            revert NotEnoughDepositETH();
        }

        _requireProtocolWithdrawalAccount(validator.withdrawalCredentials);

        usedValidators[validator.pubkey] = true;
        amountDeposited += validator.depositAmount;

        emit ValidatorInitiated({
            id: keccak256(validator.pubkey),
            operatorID: validator.operatorID,
            pubkey: validator.pubkey,
            amountDeposited: validator.depositAmount
        });
    }
```

```
        allocatedETHForDeposits -= amountDeposited;
        totalDepositedInValidators += amountDeposited;
        numInitiatedValidators += validators.length;

        // Second loop is to send the deposits to the deposit contract. Keeps external calls to the deposit contract
        // separate from state changes.
        for (uint256 i = 0; i < validators.length; ++i) {
            ValidatorParams calldata validator = validators[i];
            depositContract.deposit{value: validator.depositAmount}({
                pubkey: validator.pubkey,
                withdrawal_credentials: validator.withdrawalCredentials,
                signature: validator.signature,
                deposit_data_root: validator.depositDataRoot
            });
        }
    }
```

The check in Staking.sol on lines 425-427 should be moved to after the loop on line 441, and the check should be changed to:

```
if (amountDeposited > allocatedETHForDeposits) {
    revert NotEnoughDepositETH();
}
```

MAN1-3. REDUNDANT VARIABLE INITIALISATION

SEVERITY: Informational

PATH: see description

REMEDIATION: not initialise variables to zero, for example instead of uint256 amountDeposited = 0; we recommend using uint256 amountDeposited;

STATUS:

DESCRIPTION:

We found that at the following location in the code there is a variable initialised to its default value. This is already the default behaviour of Solidity and it becomes therefore redundant and a waste of gas, especially for storage variables.

1. Staking.sol:amountDeposited (L409)
2. Staking.sol:total (L521)
3. OracleQuorumManager.sol:relativeThresholdBasisPoints (L117)
4. UnstakeRequestsManager.sol:numCancelled (L239)

1.

```
    uint256 amountDeposited = 0;
```

2.

```
    uint256 total = 0;
```

3.

```
    relativeThresholdBasisPoints = 0;
```

4.

```
    uint256 numCancelled = 0;
```

MAN1-4. CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: Oracle.sol, OracleQuorumManager.sol, Pauser.sol, ReturnsAggregator.sol, ReturnsReceiver.sol, Staking.sol, UnstakeRequestsManager.sol.

REMEDIATION: the mentioned variables should be marked as private instead of public

STATUS:

DESCRIPTION:

In the mentioned contracts, there are constant variables that are declared **public**. However, setting constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code:

e.g.

```
/// @notice Pauser role can pause flags in the contract.
bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

/// @notice Unpauser role can unpause flags in the contract.
bytes32 public constant UNPAUSER_ROLE = keccak256("UNPAUSER_ROLE");
```

MAN1-5. UNUSED RECEIVE AND FALLBACK FUNCTION

SEVERITY: Informational

PATH: Staking.sol

REMEDIATION: remove the functions receive and fallback

STATUS:

DESCRIPTION:

In the Staking contract both **receive** and **fallback** are defined but revert when called, making them redundant.

A contract without **receive** and **fallback** will have the same functionality and cost less on deployment and for the callers.

Please note that **selfdestruct** (**sendAll** in the future) are not affected by having or not having **receive** and **fallback** and will send funds to the contract regardless.

```
receive() external payable {
    revert DoesNotReceiveETH();
}

fallback() external payable {
    revert DoesNotReceiveETH();
}
```

MAN1-14. MAGIC NUMBERS SHOULD BE REPLACED WITH CONSTANTS

SEVERITY: Informational

PATH: ReturnsAgreggator.sol:L138

REMEDIATION: change hard coded values such as 10_000 with a constant that explains the purpose of such variable, for example as _BASIS_POINTS_DENOMINATOR in ReturnsAgreggator.sol line 33

STATUS:

DESCRIPTION:

Magic numbers, e.g **10_000** hurt readability and use more gas. Replacing them with a constant like **_BASIS_POINTS_DENOMINATOR** improves readability and gas cost.

```
uint256 fees = Math.mulDiv(feesBasisPoints, totalRewards, 10_000);
```

MAN1-15. UNUSED ERRORS

SEVERITY: Informational

PATH: Oracle.sol

REMEDIATION: remove the unused errors

STATUS:

DESCRIPTION:

In the **Oracle.sol** contract the errors **InvalidUpdateStartBlock** on line 50, and **InvalidUpdate** on line 52 are declared but never used.

```
error InvalidUpdateStartBlock(uint256 wantUpdateStartBlock, uint256 gotUpdateStartBlock);
error InvalidUpdate();
```

MAN1-16. IDENTICAL FUNCTIONS

SEVERITY: Informational

PATH: Staking.sol:topUp, receiveReturns (L470-472), (L554-556)

REMEDIATION: see description

STATUS:

DESCRIPTION:

There are two functions, **topUp** and **receiveReturns**, that perform the same action of receiving funds and adding them to the unallocatedETH variable. These functions are practically identical in terms of logic, but they have different modifiers and are associated with different roles.

To enhance code clarity and reduce duplication, merge these two functions into a single function that can handle both receiving funds from the returns aggregator and performing top-up actions.

```
function topUp() external payable onlyRole(TOP_UP_ROLE) {
    unallocatedETH += msg.value;
}
```

```
function receiveReturns() external payable onlyReturnsAggregator {
    unallocatedETH += msg.value;
}
```

Change **onlyReturnsAggregator** modifier to **onlyReturnsAggregatorOrTopUpRole** which checks whether the sender is the returns aggregator or has the top-up role.

Rename the function **receiveReturns** to **receiveReturnsOrTopUp** to reflect its combined functionality.

Use the **onlyReturnsAggregatorOrTopUpRole** modifier in the **receiveReturnsOrTopUp** function.

e.g.

```
modifier onlyReturnsAggregatorOrTopUpRole() {
  if (msg.sender != returnsAggregator && !hasRole(TOP_UP_ROLE, msg.sender)) {
    revert NotAuthorized();
    }
    _;
}

function receiveReturnsOrTopUp() external payable onlyReturnsAggregatorOrTopUpRole {
  unallocatedETH += msg.value;
}
```

MAN1-20. UNSTAKE REQUEST INFO SHOULD REPORT IF THE UNSTAKE REQUEST IS FILLED

SEVERITY: Informational

PATH: UnstakeRequestsManager.sol:requestInfo

REMEDIATION: the first output of requestInfo() function should return isFinalized && isFilled instead of just isFinalized

STATUS:

DESCRIPTION:

The function **requestInfo()** reports partially claimable ether, however the **claim()** function doesn't allow claiming partially filled unstake requests, reverting when the requests isn't fully filled (L199-L201).

Therefore, it might be confusing for the user to perceive that the **requestInfo()** function reports the unstake request as finalised, but the user is still unable to claim ether.

```
  function requestInfo(uint256 requestID) external view returns (bool, uint256) {
      UnstakeRequest memory request = _unstakeRequests[requestID];

      bool isFinalized = _isFinalized(request);
      uint256 claimableAmount = 0;

      // The cumulative ETH requested also includes the ETH requested and must be subtracted from the
cumulative total
      // to find partially filled amounts.
      uint256 allocatedEthRequired = request.cumulativeETHRequested - request.ethRequested;
      if (allocatedEthRequired < allocatedETHForClaims) {
          // The allocatedETHForClaims increases over time whereas the request's cumulative ETH
requested stays the
          // same. This means the difference between the two will also increase over time. Given we only
want to
          // return the partially filled amount up to the full ETH requested, we take the minimum of the two.
          claimableAmount = Math.min(allocatedETHForClaims - allocatedEthRequired,
request.ethRequested);
      }
      return (isFinalized, claimableAmount);
  }
```

MAN1-21. ORACLE COMPATIBLILITY WITH FUTURE EIP-4788

SEVERITY: Informational

PATH: Oracle.sol

REMEDIATION: see description

STATUS:

DESCRIPTION:

Currently, the Oracle contract is quasi-decentralized, meaning there is some number of trusted Oracle reporters that should reach a quorum about the update record for the Oracle.

The next Ethereum hard-fork/upgrade Cancun https://github.com/ethereum/execution-specs/blob/master/network-upgrades/mainnet-upgrades/cancun.md anticipates to include EIP-4788 allowing to use state root for blocks from the beacon chain inside of the EVM. Therefore removing trust assumptions from Oracle reporters and allowing the Oracle contract to unequivocally verify statements about validator balances, etc. on-chain.

We would recommend to consider exploring EIP-4788 and change the Oracle implementation in such a way that Oracle reporters can submit beacon chain state root with the proof about some statement (validator balance, etc). So in the future, it will be easy to transition to a decentralized Oracle based on EIP-4788.

```
contract Oracle is Initializable, AccessControlEnumerableUpgradeable, IOracle, OracleEvents,
ProtocolEvents {
  ...
}
```