hexens × MANTLE

Sep.23

SECURITY REVIEW REPORT FOR MANTLE

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.

AUDIT LED BY

KASPER ZWIJSEN

Head of Audits | Hexens

Audit Starting Date
04.09.2023

Audit Completion Date
02.10.2023

---

hexens × MANTLE

METHODOLOGY

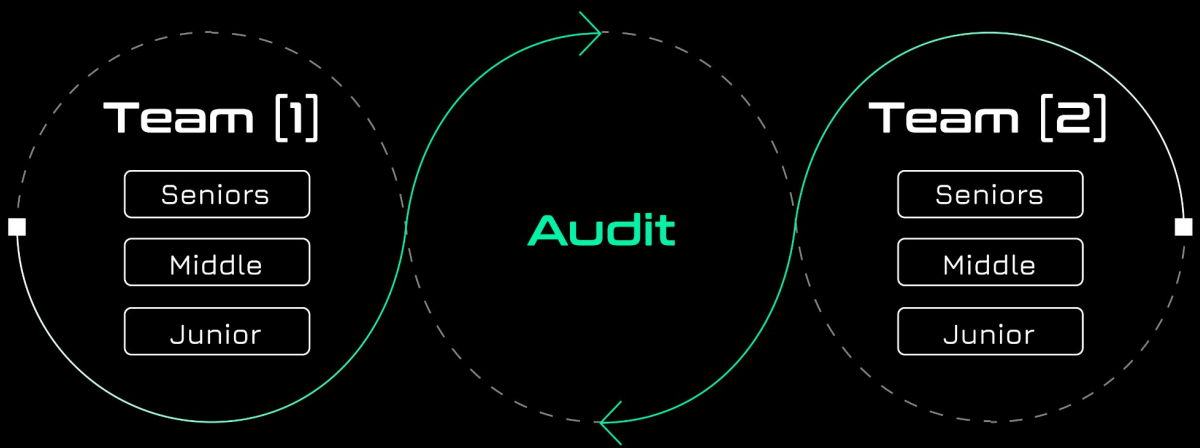
COMMON AUDIT PROCESS

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.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the oracle daemon of mETH, the new liquid staking protocol of Mantle Network. The oracle is written in Go and it is responsible for submitting oracle reports to the on-chain Oracle smart contract. Multiple trusted nodes run the same oracle software and should reach consensus on-chain.

Our security assessment was a full review of the oracle subsystem, including its libraries, spanning a total of 4 weeks.

During our audit, we have identified 3 High severity vulnerabilities. More specifically, we found edge cases where the oracle misinterpreted a validator's state and invalid reports would be generated as a result.

We have also identified various minor vulnerabilities and code quality optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality of the project have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/TwoFiftySixLabs/services/tree/commit/8e5d3c134cf71a8342bde815db59b2d7fa172b69/oracle
https://github.com/TwoFiftySixLabs/services/tree/commit/44c07aec02eaa81fab10ab20c7e88787427bc0bd/oracle
https://github.com/TwoFiftySixLabs/services/tree/commit/6624b2857a0fbdb13bc6bad2ade225e730cb8c02/oracle
https://github.com/TwoFiftySixLabs/services/tree/commit/8e5d3c134cf71a8342bde815db59b2d7fa172b69/lib
https://github.com/TwoFiftySixLabs/services/tree/commit/44c07aec02eaa81fab10ab20c7e88787427bc0bd/lib
https://github.com/TwoFiftySixLabs/services/tree/commit/6624b2857a0fbdb13bc6bad2ade225e730cb8c02/lib

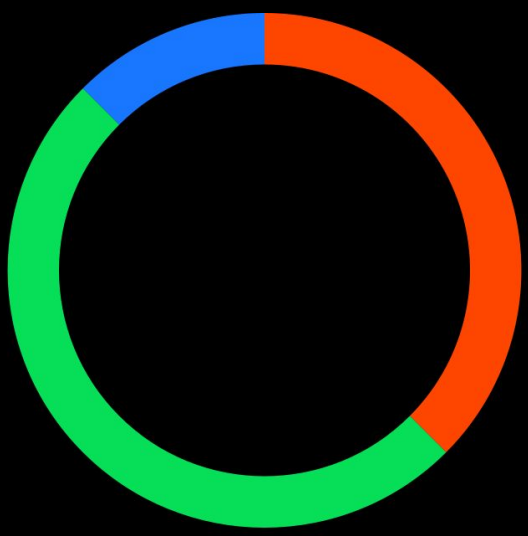
The issues described in this report were fixed in the following commit:
https://github.com/TwoFiftySixLabs/services/tree/commit/46a97ef8dbc6376b78400bbdc2d242b64c590e17

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	3
MEDIUM	0
LOW	4
INFORMATIONAL	1

TOTAL: 8

SEVERITY



● High ● Low ● Informational

STATUS

● Fixed ● Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

MAOR-1. ORACLE DOS BY DEPOSITING INTO A WITHDRAWN VALIDATOR

SEVERITY: High

PATH:
services/oracle/reporter/reporter.go:computeNumValidatorsFullyWithdrawn:L201-232

REMEDIATION: the oracle should use the validator.withdrawableEpoch timestamp to determine whether the validator is fully withdrawn. There should be no logic dependent on the balance, as this can be manipulated by anyone

STATUS: fixed

DESCRIPTION:

The oracle implementation uses a validator's balance to compute the number of validators that have withdrawn in a report period in **computeNumValidatorsFullyWithdrawn**. In the function logic, if the validator existed at the start block with a non-zero balance and has a zero balance at the end block, then the counter is increased. This record value is also cumulative, in the sense that it adds the value of the previous record to the new record.

However, this logic of determining withdrawn validators is incorrect and it can be influenced by an attacker.

The consensus layer allows for anyone to deposit ETH into any validator using the validator's public key. Only the first deposit (and the creation of the validator) requires a valid signature as proof-of-possession. This also holds for an already withdrawn validator. It won't re-activated the validator, but it will increase the balance and trigger another withdrawal.

Therefore an attacker can deposit the minimum amount of ETH into a Mantle-controlled withdrawn validator on the consensus layer. This will briefly increase the balance of the validator to non-zero and then trigger an automatic withdrawal of the full balance again.

As a result, the oracle reporter will see that for this validator, there is a positive balance in the begin block and a balance of 0 in the end block. It will count this withdrawal as a newly exited validator and increase the counter in the report, even though this validator has already been accounted for in the past.

This could cause an oracle DoS due to it becoming an invalid report and causing reverts in the sanity checker when it is submitted to the Oracle contract:

```
if (
    uint256(newRecord.currentNumValidatorsWithBalance)
        + uint256(newRecord.cumulativeNumValidatorsFullyWithdrawn) > staking.numInitiatedValidators()
) {
    revert InvalidUpdateMoreValidatorsThanInitiated(
        newRecord.currentNumValidatorsWithBalance +
newRecord.cumulativeNumValidatorsFullyWithdrawn,
        staking.numInitiatedValidators()
    );
}
```

On the other hand, if there are in-flight deposits for new validators, then staking.numInitiatedValidators would have increased and an increase of **newRecord.cumulativeNumValidatorsFullyWithdrawn** would be possible (because the sum is taken). This could make the sanity checks pass, even though this report is invalid.

```
func (r *Reporter) computeNumValidatorsFullyWithdrawn(ctx context.Context, beginVals, endVals
[]*validator.Validator) (uint64, error) {
    pubKeyToBal := make(map[string]*big.Int)
    for _, v := range beginVals {
        pubKeyToBal[v.PubKey] = v.BalanceWei
    }

    count := uint64(0)
    for _, v := range endVals {
        // If the validator still has a balance, skip it.
        if v.BalanceWei.Cmp(zero) != 0 {
            continue
        }
        // If the validator is not in the begin set and has no balance, it is added and fully withdrawn within the
current period.
        // This is very unlikely to happen as there is a 256 epoch delay for exiting after creating a validator,
but we include it
        // in case either; we have a giant report window, or the ethereum rules change.
        _, ok := pubKeyToBal[v.PubKey]
        if !ok {
            logger.Infow("Found withdrawn validator", "pubKey", v.PubKey)
            count++
            continue
        }

        prevBalance := pubKeyToBal[v.PubKey]
        // If the validator is in the begin set and has some balance, and it is fully withdrawn (i.e. no balance)
within the current period, then increase the counter.
        if prevBalance.Cmp(zero) != 0 {
            logger.Infow("Found withdrawn validator", "pubKey", v.PubKey)
            count++
        }
    }

    return count, nil
}
```

MAOR-9. VALIDATOR ISFULLYWITHDRAWN PREDICATE IS INCORRECT AND CAN CAUSE DOUBLE COUNTING

SEVERITY: High

PATH: lib/validator/validator.go:IsFullyWithdrawn:L44-46

REMEDIATION: the IsFullyWithdrawn predicate is in contradiction with the specification of the validator status. To make sure that validators cannot be counted twice, the balance comparison should be removed. If the validator has the state WithdrawalPossible and WithdrawalDone, it should be counted as fully exited in the accounting of the Oracle, regardless of the balance

STATUS: fixed

DESCRIPTION:

The newly implemented logic after validation of MAOR-1: Oracle DoS by depositing into a withdrawn validator is still vulnerable.

The reporter uses the validator predicates as filters on the total list of validators at certain points in time to determine which validator are active and which have newly exited. It is important to make sure that validators that have already been counted as exited do not get counted again, as that would invalidate the accounting in the Oracle report.

In **oracle/reporter/reporter.go**, the predicate is used against the list of validators from the end block of the previous report to obtain any validators that have already been exited and counted.

However, the predicate is incorrect:

```
func (v *Validator) IsFullyWithdrawn() bool {
    return (v.Status == v1.ValidatorStateWithdrawalPossible && v.BalanceWei.Cmp(big.NewInt(0)) == 0) ||
v.Status == v1.ValidatorStateWithdrawalDone
}
```

It marks a validator as fully withdrawn if it has the **WithdrawalPossible** state and a balance of 0. As can be seen in the specification or implementations of beacon chain clients, this state is returned whenever the effective balance is non-zero:

```
if val.WithdrawableEpoch() <= epoch {
    if val.EffectiveBalance() != 0 {
        return validator.WithdrawalPossible, nil
    } else {
        return validator.WithdrawalDone, nil
    }
}
```

https://github.com/prysmaticlabs/prysm/blob/e22258caa957fe56e35c16fd8e17a47403752b77/beacon-chain/rpc/eth/helpers/validator_status.go#L63-L69

The effective balance increases and decreases according to the actual balance and so if **WithdrawalPossible** is returned, the effective balance and balance would be non-zero, making the predicate always return false for this status due to conflicting logic.

Furthermore, it is possible to go from the **WithdrawalDone** state back to the **WithdrawalPossible** state by depositing into an already withdrawn validator. This will increase the balance and effective balance for a number of epochs, until a new withdrawal is triggered.

This is also not accounted for in the **IsFullyWithdrawn** predicate and it becomes possible to make the Oracle count exited validators multiple times. This would increase **CumulativeNumValidatorsFullyWithdrawn** and cause DoS with invalid oracle records or for invalid oracle records to be finalised in the **Oracle.sol** contract:

```
if (
    uint256(newRecord.currentNumValidatorsNotFullyWithdrawn)
        + uint256(newRecord.cumulativeNumValidatorsFullyWithdrawn) > staking.numInitiatedValidators()
) {
    revert InvalidUpdateMoreValidatorsThanInitiated(
        newRecord.currentNumValidatorsNotFullyWithdrawn +
newRecord.cumulativeNumValidatorsFullyWithdrawn,
        staking.numInitiatedValidators()
    );
}
```

https://github.com/TwoFiftySixLabs/mntEth/blob/39bbc3edafd2c5387b43c1cf331fd7e0b4802bd8/src/Oracle.sol#L355-L363

```
func (v *Validator) IsFullyWithdrawn() bool {
    return (v.Status == v1.ValidatorStateWithdrawalPossible && v.BalanceWei.Cmp(big.NewInt(0)) == 0) ||
v.Status == v1.ValidatorStateWithdrawalDone
}
```

MAOR-10. VALIDATOR'S WITHDRAWN PRINCIPAL COULD BE COUNTED AS REWARDS

SEVERITY: High

PATH:
/lib/withdrawals/withdrawals.go:ComputeWithdrawals:L57-69

REMEDIATION: in the logic, the withdrawal of the validator has already happened and so the status WithdrawalPossible is also a valid status for a fully withdrawn validator. It would mean that the withdrawable epoch has been reached and the withdrawal sweep has passed the validator, withdrawing the full principal amount

so the function should filter on validators that have either WithdrawalPossible or WithdrawalDone as status

STATUS: fixed

DESCRIPTION:

In order to correctly count withdrawals and separate principal from reward amounts, the oracle uses the **ComputeWithdrawals** function. It does so by looking at all withdrawals and filtering on Mantle validators.

The function **sumWithdrawals** then counts withdrawals from validators that have the **WithdrawalDone** state as principal (any > 32 ETH excess is counted as rewards) and other withdrawals from validators with other states are counted as rewards.

However, this logic is incorrect, as a withdrawal can happen before the validator is put into the **WithdrawalDone** state. From a beacon chain implementation (Prysm), we can see that **WithdrawalDone** is only returned when the effective balance is 0:

```
if val.WithdrawableEpoch() <= epoch {
    if val.EffectiveBalance() != 0 {
        return validator.WithdrawalPossible, nil
    } else {
        return validator.WithdrawalDone, nil
    }
}
```

https://github.com/prysmaticlabs/prysm/blob/e22258caa957fe56e35c16fd8e17a47403752b77/beacon-chain/rpc/eth/helpers/validator_status.go#L63-L69

But the effective balance of a validator is only updated at the end of an epoch. This can be seen in the consensus specifications:

```
def process_effective_balance_updates(state: BeaconState) -> None:
    # Update effective balances with hysteresis
    for index, validator in enumerate(state.validators):
        balance = state.balances[index]
        [..]
        validator.effective_balance = min(balance - balance % EFFECTIVE_BALANCE_INCREMENT,
MAX_EFFECTIVE_BALANCE)

def process_epoch(state: BeaconState) -> None:
    [..]
    process_effective_balance_updates(state)
    [..]
```

https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#effective-balances-updates

And withdrawals happen at every slot:

```
def process_withdrawals(state: BeaconState, payload: ExecutionPayload) -> None:
    [..]

def process_block(state: BeaconState, block: BeaconBlock) -> None:
    [..]
    process_withdrawals(state, block.body.execution_payload)  # [New in Capella]
    [..]
```

https://github.com/ethereum/consensus-specs/blob/dev/specs/capella/beacon-chain.md#new-process_withdrawals

And so it may happen that a validator exits and their withdrawal happens at the first slot of an epoch. The withdrawal is performed, but the validator's effective balance won't be decreased until the next epoch and so the validator's status won't go to **WithdrawalDone** for the remaining 31 slots.

This scenario could happen if an oracle report has an end block that is not the first slot of an epoch and if there are withdrawals in slots before or in the epoch of this end block.

The oracle would count this withdrawal, because it is in the slot range, but it would see the status **WithdrawalPossible** for the validator. As a result, it will count the withdrawal as a reward, as seen in **sumWithdrawals** on lines 131-135:

```
_, ok := fullyWithdrawnValidatorSet[index]
if !ok {
    reward = reward.Add(reward, withdrawalAmount)
    continue
}
```

```
func (a *Analyzer) ComputeWithdrawals(ctx context.Context, begin, end *consensus.BlockStamp,
endValidators []*validator.Validator) (principal *big.Int, reward *big.Int, err error) {
    withdrawals, err := a.findWithdrawalsInRange(ctx, begin, end)
    if err != nil {
        return nil, nil, errors.Wrap(err, "failed to find withdrawals")
    }
    // Find only withdrawals for our validators.
    ourWithdrawals := a.filterOurWithdrawals(withdrawals, valIndicesSet(endValidators))

    // Find validators which are withdrawn (i.e. have become withdrawn in this range).
    withdrawnValidators := validator.Filter(endValidators,
validator.HasState(v1.ValidatorStateWithdrawalDone))
    principal, reward = a.sumWithdrawals(ourWithdrawals, withdrawnValidators)
    return principal, reward, nil
}

func (a *Analyzer) sumWithdrawals(withdrawals []*Withdrawal, fullyWithdrawnValidators
[]*validator.Validator) (*big.Int, *big.Int) {
    principal := big.NewInt(0)
    reward := big.NewInt(0)

    // First, build a map of all validator indices in the withdrawals and use it to sum up each validator index
with its total amount
    // withdrawn in the withdrawals period.
    validatorIndexToWithdrawalAmount := make(map[uint64]*big.Int)
    for _, w := range withdrawals {
        exisingAmount, ok := validatorIndexToWithdrawalAmount[w.ValidatorIndex]
        if ok {
            validatorIndexToWithdrawalAmount[w.ValidatorIndex] = big.NewInt(0).Add(exisingAmount,
w.AmountWei)
            continue
        }
        validatorIndexToWithdrawalAmount[w.ValidatorIndex] = w.AmountWei
    }
```

```
    // Now that we have the total amount withdrawn for each validator index, we'll determine how much of
that
    // is principal and how much is a reward. To do that, we create a set of all the fully withdrawn validators
and use that to
    // decide whether it is part of the principal or a full reward.
    fullyWithdrawnValidatorSet := valIndicesSet(fullyWithdrawnValidators)
    for index, withdrawalAmount := range validatorIndexToWithdrawalAmount {
        _, ok := fullyWithdrawnValidatorSet[index]
        if !ok {
            reward = reward.Add(reward, withdrawalAmount)
            continue
        }

        if withdrawalAmount.Cmp(consensus.MaxEffectiveBalanceWei) > 0 {
            rewardAmount := big.NewInt(0).Sub(withdrawalAmount, consensus.MaxEffectiveBalanceWei)
            reward = reward.Add(reward, rewardAmount)
            principal = principal.Add(principal, consensus.MaxEffectiveBalanceWei)
            continue
        }

        principal = principal.Add(principal, withdrawalAmount)
    }

    return principal, reward
}
```

MAOR-11. ORACLE REPORT SANITY CHECK POSITIVE GAINS LIMIT COULD RESULT IN DOS

SEVERITY: Low

PATH: src/Oracle.sol:sanityCheckUpdate:L362-484

REMEDIATION: we would recommend to either remove the upper bound on the absolute value or increase the maximum gain to increase the cost for an attacker

instead, we want to recommend to implement sanity checks on changes in the share rate of mETH, which is where a potential attack is more likely to manifest. For example, if a malicious oracle report greatly increases the consensus layer balance, they would only profit if the share rate also greatly increases. However, this check should also not be too conservative, or the original issue would reappear

STATUS: acknowledged, see commentary

DESCRIPTION:

Once the oracle client has generated a report, it submits this report on-chain to the **Oracle.sol** contract. The Oracle contract will pass the report through a sanity check using **sanityCheckUpdate** and it will reject the report if it does not pass.

The sanity check contains checks on the change in the total balance of all Mantle validators on the consensus layer. Both lower and upper bounds are used.

The upper bound check is done using **maxConsensusLayerGainPerBlockPPT** and defined per block as 10 times the expected APR of 5%:

```
// 7200 slots per day * 365 days per year = 2628000 slots per year
// assuming 5% yield per year
// 5% / 2628000 = 1.9025e-8
// 1.9025e-8 per slot = 19025 PPT
maxConsensusLayerGainPerBlockPPT = 190250; // 10x approximate rate
```

Using the old (baseline) consensus layer balance, the upper bound is calculated using the report size in blocks and the maximum gain per block as defined above:

```
uint256 upperBound = baselineGrossCLBalance
    + Math.mulDiv(maxConsensusLayerGainPerBlockPPT * reportSize, baselineGrossCLBalance,
_PPT_DENOMINATOR);
if (newGrossCLBalance > upperBound) {
    return ("Consensus layer change above max gain", newGrossCLBalance, upperBound);
}
```

If the new consensus layer balance crosses this upper bound, the report is rejected.

The issue here is that this upper bound check is quite conservative and does not account for any balance manipulation that could be performed by anyone. Any user can deposit ETH into validators owned by Mantle, increasing their balance on the consensus layer and consequently the consensus layer balance that is used in the oracle report.

For example, with a report size of a 2400 slots and a total validator balance of 50,000 ETH (86.5m$) the upper bound would be calculated as a maximum increase of 22.83 ETH. The actual rewards would be about 2.283 ETH, so the attacker would have to supply 20.55 ETH.

The cost of this attack is quite high but it might still become an attack vector for large competitors that want to disrupt Mantle's protocol.

```
function sanityCheckUpdate(OracleRecord memory prevRecord, OracleRecord calldata newRecord)
    public
    view
    returns (string memory, uint256, uint256)
{
    uint64 reportSize = newRecord.updateEndBlock - newRecord.updateStartBlock + 1;
    [..]
    {
        //
        // 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 * reportSize, 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 * reportSize, baselineGrossCLBalance,
_PPT_DENOMINATOR);

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

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

Commentary from the client:

“ - Protocol is easily recoverable and the bounds are settable if this were to ever happen."

MAOR-4. LACK OF VALIDATION OF EXECUTION_OPTIMISTIC PARAMETER

SEVERITY: Low

PATH: see description

REMEDIATION: to remediate the issue, the execution_optimistic parameter should be checked. If it contains a true value, the response from the Quicknode API should be discarded

STATUS: acknowledged, see commentary

DESCRIPTION:

The Oracle service in the project utilizes the Quicknode API to obtain information about validators from the beacon chain. This information is then used to generate a report within the **BuildOracleReport()** function in **reporter.go**. However, there is an issue in which the Oracle service does not validate the execution_optimistic boolean parameter returned by the Quicknode API. For more details, please refer to the eth/v1/beacon/states/{state_id}/validators RPC Method | Ethereum Documentation. This parameter indicates that the information retrieved may be optimistic and subject to invalidation at a later time. As a result, the report generated by the Oracle service may contain incorrect data.

Currently, the flow of building the Oracle report is as follows:

1. The **BuildOracleReport()** function in **reporter.go** calls **ValidatorsAt()** to get validators' information for a specific slot ID.
2. **ValidatorsAt()** internally calls **getConsensusValidatorsAt()** of **ParallelLoader**.

3. **getConsensusValidatorsAt()** subsequently calls **ValidatorsByPubKey()** of **consensus.Client**, which is based on **attestantio/go-eth2-client**.
4. It makes a GET request to the **/eth/v1/beacon/states/{state_id}/validators** endpoint of the RPC node, which is configured from the **beacon-node-url** parameter or the **BEACON_NODE_URL** environment variable.
5. Finally, the **.env.goerli** file in the repository points to a node URL of Quicknode API.

```
BEACON_NODE_URL=https://responsive-twilight-mountain.ethereum-goerli.discover.quiknode.pro/3e30bbff60394e2cde1dce177b52321b3625c4be/
```

```
func (l *ParallelLoader) getConsensusValidatorsAt(ctx context.Context, pubkeys []string, bs *consensus.BlockStamp) ([]*v1.Validator, error) {
    logger.Infow("Getting all our validators", "BlockStamp", bs, "count", len(pubkeys))
    metrics.Count("sourcer_load_validators", float64(len(pubkeys)))

    pubkeyFilters, err := consensus.CreatePubKeyFilters(pubkeys, maxPubkeysInFilter)
    if err != nil {
        return nil, errors.Wrap(err, "failed to create pubkey filter")
    }

    tasks := make([]runner.TaskFunc[[]*v1.Validator], len(pubkeyFilters))

    for i, pkf := range pubkeyFilters {
        // We need to capture the value of i and pkf in the closure, so we create a new variable
        // for each iteration.
        i, pkf := i, pkf
        tasks[i] = func(ctx context.Context) ([]*v1.Validator, error) {
            // There are no guarantees for the returned data in terms of ordering.
            // https://ethereum.github.io/beacon-APIs/#/Beacon/getStateValidators
            // We search by SlotNumber as there is an issue with the devnet where fetching validators
```

```
        // fails for a "head" state root. Both are equivalent values to search for according to the
documentation.
        //
https://github.com/attestantio/go-eth2-client/blob/bbf582d19f7ede31e7d1a2492e76539c3926d457/http/v
alidatorsbypubkey.go#L70-L73
        valMap, err := l.consensusClient.ValidatorsByPubKey(ctx, fmt.Sprintf("%d", bs.SlotNumber), pkf)
        if err != nil {
            return nil, errors.Wrapf(err, "failed to get validators by pubkey %s", pkf)
        }
        vals := validatorMapToSlice(valMap)

        return vals, nil
      }
    }

    limiter := rate.NewLimiter(rate.Limit(l.maxRPS), l.maxRPS)
    results, err := runner.ExecuteTasks[[]*v1.Validator](ctx, limiter, tasks)
    if err != nil {
        return nil, err
    }

    var vals []*v1.Validator
    for _, res := range results {
        vals = append(vals, res...)
    }
    return vals, nil
}
```

Commentary from the client:

“ - We are working on a fix to verify using these fields, but it requires coordinating upstream changes with our eth2 client."

MAOR-7. NODE API DEPENDENCY FOR VERIFICATION OF VERSIONEDSIGNEDBEACONBLOCK

SEVERITY: Low

PATH: see description

REMEDIATION: it would be infeasible for the oracle client to fully validate all data in a beacon block, as this would require it to validate the whole blockchain and almost turning it into a full Ethereum node

instead, it is crucial that the oracle client only relies on Mantle's own or fully-trusted Ethereum nodes as API. Preferably multiple nodes as well so the oracle client can validate that the output of all nodes match

STATUS: fixed, see commentary

DESCRIPTION:

The oracle service retrieves a signed beacon block without verifying the data or signatures. The node API it currently uses is the QuickNode API. The issue exists in the following functions: **ComputeWithdrawals()** called from **OracleReporter** class (line 162), and **alignBlockWindow()** in the Scheduler class (lines 311 and 356).

The consensus client in use is based on the **attestantio/go-eth2-client** library, which also lacks any verification within the **SignedBeaconBlock()** function:
https://github.com/attestantio/go-eth2-client/blob/bbf582d19f7ede31e7d1a2492e76539c3926d457/http/signedbeaconblock.go#L78-L127

Consequently, if the returned **VersionedSignedBeaconBlock** from **SignedBeaconBlock()** contains forged data, the oracle service may report incorrect information.

This is currently under complete control of the node API and so it creates a risk of dependency on this node.

```
func (a *Analyzer) findWithdrawalsAtSlot(ctx context.Context, slotNumber uint64) ([]*Withdrawal, error) {
    block, err := a.consensusClient.SignedBeaconBlock(ctx, strconv.FormatUint(slotNumber, 10))
    [..]
}
```

```
func (s *Scheduler) alignBlockWindow(ctx context.Context, lastRecord oracle.OracleRecord) (uint64,
uint64, error) {
    // Find the latest finalized block
    block, err := s.consensusClient.SignedBeaconBlock(ctx, consensus.BlockIdentifierFinalized)
}
```

Commentary from the client:

“ - To mitigate we are running multiple oracles with multiple data sources, so any discrepancy or attack should cause the oracle not to reach quorum before it causes an issue."

MAOR-12. BLOCKSTAMP LIBRARY USES PARENT HASH INSTEAD OF BLOCK HASH

SEVERITY: Low

PATH: /lib/consensus/blockstamp.go:L64, L72

REMEDIATION: use BlockHash instead of ParentHash to construct the BlockStamp struct

STATUS: fixed

DESCRIPTION:

The library for creating a BlockStamp from a beacon block, exposes the function NewFromSignedBeaconBlock to facilitate this.

However, for both Capella and Deneb versions, it will place the beacon block's ParentHash in the BlockHash, instead of the beacon block's BlockHash:

```
return &BlockStamp{
    StateRoot:     root.String(),
    SlotNumber:    uint64(slot),
    BlockNumber:   block.Capella.Message.Body.ExecutionPayload.BlockNumber,
>>>>        BlockHash:     block.Capella.Message.Body.ExecutionPayload.ParentHash.String(),
    BlockTimestamp: block.Capella.Message.Body.ExecutionPayload.Timestamp,
}, nil
```

This field is currently not used anywhere, so the impact is very low. But it were ever to be used, it could immediately result in an impactful bug.

```
func NewFromSignedBeaconBlock(block *spec.VersionedSignedBeaconBlock) (*BlockStamp, error) {
    root, err := block.StateRoot()
    if err != nil {
        return nil, err
    }
    slot, err := block.Slot()
    if err != nil {
        return nil, err
    }
    switch block.Version {
    case spec.DataVersionBellatrix:
        return &BlockStamp{
            StateRoot:      root.String(),
            SlotNumber:     uint64(slot),
            BlockNumber:    block.Bellatrix.Message.Body.ExecutionPayload.BlockNumber,
            BlockHash:      block.Bellatrix.Message.Body.ExecutionPayload.BlockHash.String(),
            BlockTimestamp: block.Bellatrix.Message.Body.ExecutionPayload.Timestamp,
        }, nil
    case spec.DataVersionCapella:
        return &BlockStamp{
            StateRoot:      root.String(),
            SlotNumber:     uint64(slot),
            BlockNumber:    block.Capella.Message.Body.ExecutionPayload.BlockNumber,
            BlockHash:      block.Capella.Message.Body.ExecutionPayload.ParentHash.String(),
            BlockTimestamp: block.Capella.Message.Body.ExecutionPayload.Timestamp,
        }, nil
    case spec.DataVersionDeneb:
        return &BlockStamp{
            StateRoot:      root.String(),
            SlotNumber:     uint64(slot),
            BlockNumber:    block.Deneb.Message.Body.ExecutionPayload.BlockNumber,
            BlockHash:      block.Deneb.Message.Body.ExecutionPayload.ParentHash.String(),
            BlockTimestamp: block.Deneb.Message.Body.ExecutionPayload.Timestamp,
        }, nil
    }
    return nil, fmt.Errorf("unknown block version %d", block.Version)
}
```

MAOR-5. CONSIDER ADDING WEBSOCKET SUPPORT FOR THE CONSENSUS LAYER RPC CLIENT

SEVERITY: Informational

REMEDIATION: see description

STATUS: acknowledged, see commentary

DESCRIPTION:

Currently, the consensus layer RPC client only supports the HTTP protocol. The URL for the client is configured using the **beacon-node-url** parameter in the configuration file or command line parameter.

In contrast, the execution layer RPC client supports both the WebSocket and HTTP protocols. The URL for this client is configured using the **etherrpc-url** parameter in the configuration file. This is possible because the **go-ethereum/ethclient** library, which is used as the execution layer RPC client, has built-in support for both protocols.

Commentary from the client:

“ - We have not seen any performance issues yet but will keep this in mind for the future."

```go
// NewClients creates clients that the service needs.
func NewClients(ctx context.Context, conf *Config) (*Clients, error) {
    consensusClient, err := consensus.NewClient(ctx, conf.BeaconNodeURL, conf.BeaconAuthToken)
    if err != nil {
        return nil, errors.Wrap(err, "failed to create consensus client")
    }

    ethClient, err := ethclient.Dial(conf.RPCUrl)
    if err != nil {
        return nil, errors.Wrap(err, "failed to create execution client")
    }

    return &Clients{
        Execution: ethClient,
        Consensus: consensusClient,
    }, nil
}
```