PRIVATE

Code Assessment

of the Staking Contracts

Smart Contracts

March 20, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Polygon with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of the staking contracts according to Scope to support you in forming an opinion on their security risks.

This review covers the staking contracts for Polygon's PoS network. These contracts govern the staking of MATIC tokens for validators, including verifying validators signatures on checkpoints. An auxilary contract implements functionality for Delegators to delegate stake to validators.

The most critical subjects covered in our audit are asset security and solvency, functional correctness and signature verification. Ascertaining the functional correctness was challenging, since no detailed specification was available. Generally, assessing the solvency of the contract is challenging given that the architecture doesn't allow to distinguish which part of the MATIC balance held belongs to stake, rewards or fees.

An rewards-related issue was successfully resolved. For most other issues Polygon responded with "Risk accepted" without providing additional information. Overall we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Medium-Severity Findings	1
• Risk Accepted	1
Low-Severity Findings	9
• Code Corrected	1
• Risk Accepted	7
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Staking Contracts repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	27 September 2022	5dace23fd378b89aef3ef9ad6407cc44eac4d7db	Initial Version
2	27 February 2023	1eb6960e511a967c15d4936904570a890d134fa6	After Intermediate Report

For the solidity smart contracts, the compiler version `0.5.17` was chosen.

The StakeManager and ValidatorShare folders in `/staking` as well as `Eventshub.sol`, `EventsHubProxy.sol` and `StakingInfo.sol` are in scope of this review. Any functionality/code related to auctions or slashing, even inside the reviewed contracts, is not in scope and has not been reviewed. Accordingly, the exchange rate in ValidatorShare is assumed to be 1:1. Consequences should this assumption break have not been investigated.

2.1.1 Excluded from scope

- All files outside of `/staking`
- `staking/slashing`

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Polygon's Proof of Stake network (Polygon PoS) works with smart contracts on Ethereum for staking management. Stakers of MATIC tokens (Validators) sign on checkpoints of Polygon PoS. The RootChain contract submits checkpoints and signatures to the StakeManager contract for verification. This verification succeeds when at least 2/3+1 of the total active stake power have signed the checkpoint. Afterwards, the successful verification rewards are distributed to the active validators.

At the core, this is managed by the StakeManager contract, which is deployed behind an upgradeable proxy (StakeManagerProxy).

This contract exposes the `checkSignatures()` function used by the RootChain contract. Users can stake/unstake MATIC tokens in order to participate as a validator in the consensus. A validator's task is to run a full node, produce and validate blocks. Based on this, a validator can create/sign checkpoints over a set of blocks.

Active validators signing on a checkpoint are eligible for a reward in that epoch. The total reward of the epoch to be distributed depends on the amount of blocks in the checkpoints. If there are more blocks than expected, the reward is reduced for the extra blocks. The reduction increases with more full intervals and there is a limit to how many intervals can be rewarded at all. The reward is also proportional to the percentage of stake power signing on that checkpoint. This is done to incentivize reaching the target amount of blocks within a checkpoint and so that the proposer is rewarded for submitting all signatures (without omissions).

From this total reward, a part is given directly to the validator proposing the checkpoint. The remainder is distributed proportionally between the validators having signed on the checkpoint according to their total stake at this time. This reward is not immediately assigned. Instead, a global `rewardPerStake` variable is updated. The individual reward of a validator is only evaluated and updated later, which must be done before any change in the validators state.

Reward is paid out in MATIC tokens. **These MATIC tokens need to be provided to the contract. The staking contract holds a significant amount of MATIC tokens (most being stake of the Validators). It cannot distinguish between MATIC tokens belonging to stake or reward. If no or insufficient additional MATIC tokens are provided to the contract, reward may be paid out using the staked tokens of the validators.** Additionally, the StakeManager holds the MATIC tokens for the transaction fees on heimdall, this amount is negligibly small compared to stake and rewards.

The amount of active validators is limited to `validatorThreshold`, which is currently set to 100.

A validator position is identified and owned by a transferrable NFT. The owner of the NFT has full access to the validator position, e.g. can update the signer address, withdraw rewards or unstake. This NFT is minted when staking. After unstaking, there is a cooldown period until the stake can be withdrawn. During this withdrawal, the NFT is burned.

While the code features functionality to support slashing and auctions for validator slots, this functionality is currently inactive and has not been reviewed.

- Auctions are currently inhibited until epoch 2018083 (value of `replacementCoolDown`). This prevents starting a new auction before this epoch. If needed, this value can be increased by the governance.
- Slashing is technically not inhibited but assumed not to be done. 2/3+1 of the validators (based on stake) may trigger a slashing. The SlashingManager contract is deployed and registered in the system.

Users wishing to stake MATIC tokens on behalf of a validator can do so by delegation. If a validator accepts delegation, a ValidatorShareProxy contract is deployed. This proxy loads the address of the implementation code that is executed from a central Polygon registry contract.

Users can buy/sell vouchers in this ValidatorShare contract, which means staking/unstaking MATIC tokens on behalf of a Validator. As representation of their position, users get pool share tokens. These are transferrable ERC20 tokens, however their `approve()` function has been disabled. Based on the share amount, a users earns a reward. This reward originates from the delegated stake of the validator in the StakeManager, of which the validator may deduct a commission up to 100%. Selling shares is subject to an unbonding period. The tokens may only be withdrawn after the unbonding period ends.

Through the StakeManager, a user can seamlessly migrate his delegation from one Validator to another without being subject to an unbonding period.

Due to the inhibited slashing, the exchange rate between validator shares and delegate stake is 1:1. Similarly, this holds for the withdrawExchangeRate.

Furthermore, there are the contracts EventHub and StakingInfo used to emit events. This enables monitoring of these contracts only to catch all events emitted by contracts of the system.

2.3 Trust Model & Roles

All contracts are upgradeable.

Within the StakeManager, the Governance has privileges including but not limited to:

- Force unstake any validator immediately
- Change the current epoch arbitrarily
- Change the staking token
- Reinitialize the contract
- Drain all staked tokens

The Governance is trusted to act honestly and correctly at all times. Incorrect actions may break the system.

Validators and Delegators are generally untrusted.

Staking token: Expected to be the MATIC token only

Other system contracts such as the RootChainContract or the Registry: Fully trusted.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- No Separation of Stake and Reward Risk Accepted

Medium-Severity Findings	1

- maxRewardedCheckpoints Not Respected Risk Accepted

Low-Severity Findings	8

- Confusing Naming Risk Accepted
- ForceUnstake Can Unstake Multiple Times Risk Accepted
- Incorrect Documentation Risk Accepted
- Ongoing Exit and _buyShares() Risk Accepted
- Outdated Compiler Version Risk Accepted
- Unnecessary getOrCacheEventsHub() Acknowledged
- _liquidateRewards() Updates initialRewardPerStake Risk Accepted
- totalRewardsLiquidated May Be Inaccurate Risk Accepted

5.1 No Separation of Stake and Reward

Design High Version 1 Risk Accepted

Validators lock MATIC tokens as stake. Upon performing their duties correctly, validators receive rewards in the form of MATIC tokens. Withdrawing stake and claiming rewards is done using the same `_transferToken()` function to transfer the specified amount of MATIC token. Furthermore, the ValidatorShare contracts can simply call function the `transferFunds()` to transfer MATIC tokens to the Delegators.

There is no separation between stake and rewards. Most notably, if no or insufficient rewards have been provided, rewards are being paid out from stake. The design of the system is fragile. Any error in the calculation / payout of rewards in the code of the StakingManager or ValidatorShare contract may transfer MATIC tokens supposed to be locked as stake. An example is issue Unstaked validators continue receiving rewards without signing .

The issue is amplified:

- It's not documented that the MATIC for the rewards need to be supplied to the contract.

- Monitoring or assessing the contract's health is difficult, see separate issue (Assessing the financial health of the StakeManager) for a detailed explanation.

Risk accepted:

Polygon has acknowledged the identified risk and, without providing additional information, has chosen to accept it.

5.2 `maxRewardedCheckpoints` Not Respected

Correctness | Medium | Version 1 | Risk Accepted

Calculating the total reward amount for a checkpoint is done in `StakeManager._calculateCheckpointReward()`. The code contains some comments roughly explaining the idea, but no detailed specification is available.

At the beginning of the function, the amount of `fullIntervals` is calculated by dividing the argument `blockInterval` by the `targetBlockInterval` per checkpoint. The result is bounded to `maxRewardedCheckpoints`, which is currently set to 3.

Later, the actual reward amount is calculated. This calculation can award rewards for more blocks than `maxRewardedCheckpoints`.

```
if (blockInterval > targetBlockInterval) {
    // count how many full intervals
    uint256 _rewardDecreasePerCheckpoint = rewardDecreasePerCheckpoint;

    // calculate reward for full intervals
    reward = ckpReward.mul(fullIntervals).sub(
        ckpReward.mul(((fullIntervals - 1) * fullIntervals / 2).mul(_rewardDecreasePerCheckpoint)).div(CHK_REWARD_PRECISION));

    // adjust block interval, in case last interval is not full
    blockInterval = blockInterval.sub(fullIntervals.mul(targetBlockInterval));
    // adjust checkpoint reward by the amount it suppose to decrease
    ckpReward = ckpReward.sub(ckpReward.mul(fullIntervals).mul(_rewardDecreasePerCheckpoint).div(CHK_REWARD_PRECISION));
}

// give proportionally less for the rest
reward = reward.add(blockInterval.mul(ckpReward).div(targetBlockInterval));
```

In case `fullInterval` has been bounded to `maxRewardedCheckpoints` but the actual value would be higher, the calculation

```
/ adjust block interval, in case last interval is not full
blockInterval = blockInterval.sub(fullIntervals.mul(targetBlockInterval));
```

will return the number of blocks in the given `blockInterval` minus `maxRewardedCheckpoints` times `targetBlockInterval`. However, as the actual amount of full intervalls exceeds `fullIntervals` used in this calculation, the amounts of blocks left can exceed `targetBlockInterval` and hence can span across two or more intervals. In this scenario, the comment "adjust block interval in case last interval is not full" is incorrect / the result of the calculation is not as expected within [1, `targetBlockInterval`].

Based on this amount of blocks, the final amount of rewards to pay is calculated:

```
reward = reward.add(blockInterval.mul(ckpReward).div(targetBlockInterval));
```

The variable name and the comment contradict the current behavior of the code.

Risk accepted:

Polygon has acknowledged the identified risk and, without providing additional information, has chosen to accept it.

5.3 Confusing Naming

Design Low Version 1 Risk Accepted

Misleading variable or function names make it hard to understand the code and may lead to errors.

In ValidatorShare, the mapping which tracks the last `RewardPerShare` value the user got rewards for has the name `initalRewardPerShare`. However, it does not only store the initial value upon share creation. It is continuously updated. This may be confusing when trying to understand the system.

In StakeManager, `initialRewardPerStake` may be similarly confusing.

Function `getTotalStake(address user)` in ValidatorShare may be misunderstood to return the totalStake of all users, while it actually returns this user's stake and the current exchange rate.

5.4 ForceUnstake Can Unstake Multiple Times

Design Low Version 1 Risk Accepted

The `onlyGovernance` function `forceUnstake()` has no check that prevents the unstaking of a validator that has already been unstaked. If a validator is unstaked multiple times, this will bring the system into an inconsistent state.

The non-privileged `unstake()` function explicitly disallows unstaking multiple times by requiring:

```
validators[validatorId].deactivationEpoch == 0
```

Governance should be very careful when using `forceUnstake()` and must ensure it does not use it on a `validatorId` that has been previously unstaked.

5.5 Incorrect Documentation

Correctness Low Version 1 Risk Accepted

The documentation contains several inaccurate statements:

https://wiki.polygon.technology/docs/maintain/validator/core-components/staking

```
Validators in Polygon Network are selected via an on-chain auction
process which happens at regular intervals.
```

The auction process is disabled. Validators can join when there is a free spot (currently all slots are in use).

```
Validators can add more MATIC tokens to their stake:

* To earn more rewards.
* To maintain the position in the validator set.
```

The second point does not really apply since auctions are disabled.

https://wiki.polygon.technology/docs/pos/contracts/stakingmanager/

- Section Validator Replacement does not apply since auctions are disabled.
- There's only function `stakeFor`, function `stake` does not exist in the reviewed implementation of the StakingManager.

https://wiki.polygon.technology/docs/maintain/validator/rewards/#transaction-fees

```
Each block producer at Bor is given a certain percentage of the
transaction fees collected in each block. The selection of producers
for any given span is also dependent on the validator’s ratio in the
overall stake. The remaining transaction fees flow through the same
funnel as the rewards which get shared among all validators working
at the Heimdall layer.
```

Since eip1559 has been implemented in Bor, all fees go to the block producer and the rest are burnt. The last sentence above no longer applies.

https://wiki.polygon.technology/docs/pos/contracts/delegation/#overview

- This description of the Validator Shares seems outdated. For example, in the reviewed implementation of the ValidatorShare, rewards work differently. They are not inclduded in the exchange rate. Hence, the initial paragraph describing MATIC/VATIC is inaccurate.
- `updateCommissionRate()` and `updateReward()` are not part of the reviewed ValidatorShare implementation.
- The "new" functions for sell / unclaim validator shares, based on nonces, are not documented.

Risk accepted:

While the documentation has been updated since the main review took place, only the section Validator Replacement of the staking manager has been corrected. All other highlighted discrepancies still apply.

5.6 Ongoing Exit and `_buyShares()`

Correctness | Low | Version 1 | Risk Accepted

`_buyShares()` reverts if the Delegator has an ongoing exit:

```
require(unbonds[user].shares == 0, "Ongoing exit");
```

This only takes into account exits started through `sellVoucher()` not yet claimed using `unstakeClaimTokens()`. Ongoing exits through `sellVoucher_new()` which are tracked in the mapping
`mapping(address => mapping(uint256 => DelegatorUnbond)) public unbonds_new;`
are not taken into account and hence do not prevent buying new shares. This behavior is inconsistent.

5.7 Outdated Compiler Version

Design Low Version 1 Risk Accepted

The project's truffle config specifies an outdated version of the Solidity compiler.

```
solc: {
   version: '0.5.17',
```

Albeit 0.5.17 being the latest 0.5.x release, there are some known bugs:

https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1207

More information about these bugs can be found here: https://docs.soliditylang.org/en/latest/bugs.html

At the time of writing, the most recent Solidity release is version `0.8.17`. Upgrading to a more recent solidity version with breaking changes must be done with great care.

5.8 Unnecessary `getOrCacheEventsHub()`

Design Low Version 1 Acknowledged

The ValidatorShares contract has a function `getOrCacheEventsHub()`:

```
function _getOrCacheEventsHub() private returns (EventsHub) {
    EventsHub _eventsHub = eventsHub;
    if (_eventsHub == EventsHub(0x0)) {
        _eventsHub = EventsHub(Registry(stakeManager.getRegistry()).contractMap(keccak256("eventsHub")));
        eventsHub = _eventsHub;
    }
    return _eventsHub;
}
```

This function is used to access the EventsHub address. It is called in `initialize()`. At this point the EventsHub address is queried from the registry and stored. Afterwards, since the address is stored, it is simply returned. Overall this function seems unnecessary, the eventsHub could simply be queried and set in `initialize` and the variable be used directly afterwards. This would be more gas-efficient.

5.9 `_liquidateRewards()` Updates `initialRewardPerStake`

Correctness Low Version 1 Risk Accepted

In `_liquidateRewards()`, the validator's `initialRewardPerStake` variable is updated. `initialRewardPerStake` keeps track of the `rewardPerStake` value up to which the validator has already received their rewards.

`_updateRewards()` updates the reward / delegatorsReward based on the increase of `rewardPerStake` since the last update.

`_liquidateRewards()` simply pays out the accumulated reward of the validator. There is no reason for it to interact with/update the validator's `initialRewardPerStake`, which is done in `_updateRewards`.

The update of the value does not lead to problems in practice, as `_liquidateRewards` is almost always called after `_updateRewards()` which updates `initialRewardPerStake` and hence the second update in `_liquidateRewards()` is a no-op.

The only exception is in `unstakeClaim()`, where `_liquidateRewards` is called by itself. After claiming their stake, the validator shouldn't receive any more rewards anyway, so this is not a problem. It does lead to an unexpected value of `initialRewardPerStake`, which makes it look like the last rewards for the validator were claimed at a later time than they actually were.

5.10 `totalRewardsLiquidated` May Be Inaccurate

Correctness | Low | Version 1 | Risk Accepted

`StakeManager.totalRewardsLiquidated` keeps track of all rewards paid out in `_liquidateRewards()`. There is no detailed specification on this variable and what exactly it keeps track of.

Using function `StakeManager.restake()`, accrued rewards may be restaked immediately without the tokens being withdrawn, hence without function `_liquidateRewards()` being invoked nor the value of `totalRewardsLiquidated` being updated.

After unstaking and later withdrawing the stake, this reward has essentially been withdrawn.

Hence, the total amount of rewards withdrawn may exceed the amount tracked in `totalRewardsLiquidated`. Due to missing specification, it's unclear if this is intended or a bug. If this is intentional, this behavior should be clearly documented.

A very similar issue exists in the ValidatorShares implementation with event `logDelegatorClaimRewards`. For rewards restaked using `restake()`, the event is not emitted and hence is unsuitable to track the amount of all paid rewards.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Unstaked Validators Continue Receiving Rewards Without Signing Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	1

- _increaseRewardAndAssertConsensus() Comment Incorrect Code Corrected

6.1 Unstaked Validators Continue Receiving Rewards Without Signing

Correctness High Version 1 Code Corrected

In StakeManager, `_increaseRewardAndAssertConsensus()` increases the rewards. It avoids giving rewards to validators that have not signed, by explicitly updating the `initialRewardPerStake` variable for all validators that are included in the `signers` array but have not signed.

However, it does not avoid giving rewards to validators that are not in the `signers` array and have not signed. This can happen when a validator unstakes. `_unstake()` will remove the validator's signer from the `signers` array. After this, the validator will always receive rewards even without signing. If they never call `unstakeClaim()`, they and their delegators will continue receiving rewards for their combined staked amount. If they do call `unstakeClaim()`, the validator's amount will be zero, so they will no longer receive any rewards. However, their delegators will continue receiving rewards and will also still be able to claim them.

The reward amount for all validators is calculated based on the signedStakePower. It is expected that only those that have signed will receive the reward. Since that assumption is broken here, the total distributed staking rewards will be higher than intended (as if the unstaked validator had also signed).

As soon as `exitEpoch` passes, the stake of the validator is removed from the `validatorState.amount`. This will lead to rewards being even more wrong. All validators will receive more rewards, since the total staked amount is decreased. In particular, this also leads to the unstaking validator receiving more rewards than if they were active.

Consider the following situation to illustrate a possible attack:

The `validatorThreshold` is 2. There are 2 validators with 5 million MATIC each. The total annual reward among all stakers is supposed to be 1 million MATIC.

One of the validators does the following:

1. `unstake()`
2. After `updateTimeline()` is called for the `exitEpoch`, `stakeFor(1 MATIC)` so that nobody else can stake.

3. Never call `unstakeClaim()` and keep earning rewards.

Now `validatorState.amount` is only 5 million + 1 MATIC, leading to rewards per MATIC staked being twice as high. Since the unstaked validator also keeps earning rewards, the total annual staking reward is now 2 million MATIC. The total staking rewards are higher than they should be.

Incorrect total staking rewards can also happen accidentally, without a deliberate attack. Consider the following:

The `validatorThreshold` is 100. There are 100 validators with 1 million self-stake and 1 million delegation each. The total annual reward among all stakers is supposed to be 20 million MATIC.

1. A staker is removed from the validator set using `_unstake()` and is replaced with a new one.
2. The staker calls `unstakeClaim()` to receive back his stake.
3. The `delegatedAmount` is unchanged and the delegators continue receiving rewards.
4. Delegators can claim their rewards using `withdrawRewards()` or `sellVoucher()`.

Now there are 201 million MATIC earning rewards, but `validatorState.amount` is only 200 million. The total annual staking reward is now 20.1 million MATIC.

Staking rewards will be exhausted sooner than they should be. If the StakingManager does not hold a sufficient buffer of tokens to cover the extra rewards, it may become insolvent and unable to let all stakers redeem their staked tokens.

6.1.1 On-chain evidence

We have found on-chain evidence of the second example happening in practice:

The validator with `validatorId 6` was unstaked in block `12246720` on Apr 15 2021. The user `0xce424875b0d20bc615c9a85aaabb83c642f311dc` (https://etherscan.io/address/0xce424875b0d20bc615c9a85aaabb83c642f311dc?toaddress=0x610AEBd620437B4b4e88801abD0c8BA0D009DF5C) had delegated to validator 6 before it was unstaked. 8 days after the unstaking happened, the user called `withdrawRewards()` (https://etherscan.io/tx/0x3a7a76bdbb52298e66254c384de0795fff5c1903c559feefbd4bafd329866b30) to receive all his accrued rewards. Since the validator is unstaked, the user should no longer be able to accrue any more rewards. However, when another 8 days later the user callls `sellVoucher_new()`, (https://etherscan.io/tx/0x3a7a76bdbb52298e66254c384de0795fff5c1903c559feefbd4bafd329866b30) they claim an additional 241 MATIC tokens. These are rewards that accrued after the validator was unstaked and without the validator signing any checkpoints.

Code corrected:

The matter was resolved by taking the following measures: Upon a validator's unstaking, the growth of Validator and Delegator rewards is halted.

`_updateRewardsAndCommit()` now features an appropriate check and returns without update for unstaked validators:

```
function _updateRewardsAndCommit(
    uint256 validatorId,
    uint256 currentRewardPerStake,
    uint256 newRewardPerStake
) private {
    uint256 deactivationEpoch = validators[validatorId].deactivationEpoch;
    if (deactivationEpoch != 0 && currentEpoch >= deactivationEpoch) {
```

```
            return;
        }
```

New delegator rewards are only evaluated for validators which have not unstaked:

```
function delegatorsReward(uint256 validatorId) public view returns (uint256) {
    uint256 _delegatorsReward;
    if (validators[validatorId].deactivationEpoch == 0) {
        (, _delegatorsReward) = _evaluateValidatorAndDelegationReward(validatorId);
    }
    return validators[validatorId].delegatorsReward.add(_delegatorsReward).sub(INITIALIZED_AMOUNT);
}
```

Note that already accrued delegatorRewards are still considered and hence claiming rewards accrued before unstaking is still possible.

6.2 `_increaseRewardAndAssertConsensus()` Comment Incorrect

Correctness **Low** **Version 1** **Code Corrected**

`_increaseRewardAndAssertConsensus()` contains the following:

```
// evaluate rewards for validator who did't sign and set latest reward per stake to new value
// to avoid them from getting new rewards.
_updateValidatorsRewards(unsignedValidators, totalUnsignedValidators, newRewardPerStake);

// distribute rewards between signed validators
rewardPerStake = newRewardPerStake;

// evaluate rewards for unstaked validators to avoid getting new rewards until they claim their stake
_updateValidatorsRewards(deactivatedValidators, totalDeactivatedValidators, newRewardPerStake);
```

The third comment is incorrect. Rewards for unstaked validators are not avoided, they are paid out. They would be avoided if `_updateValidatorsRewards()` was called before updating `rewardPerStake`, as is done for the `unsignedValidators`.

The code is correct. Since the validator should receive rewards for signing during the epoch it started unstaking, it is correct to explicitly give it rewards. This is also required, because the validator should not be allowed to receive any more rewards after this explicit last reward.

The comment should reflect that the validator explicitly receives one last reward for signing during the unsigning epoch and that it will be unable to receive more rewards later.

Code corrected:

The comment has been changed to:

```
// evaluate rewards for unstaked validators to ensure they get the reward
for signing during their deactivationEpoch
```

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Assessing the Financial Health of the StakeManager

Note Version 1

The staking contract holds a significant amount of MATIC tokens. This amount must cover all its obligations.

The tokens held by the contract fall into the following categories:

1. Validator self-stake
2. Delegated stake
3. Deactivated stake that is unbonding or unclaimed
4. Accrued Heimdall fees
5. Distributed rewards waiting to be claimed
6. Buffer of tokens for future rewards

All these categories are held in a single contract, they are not separated. If one of them is affected by a bug, the others may be affected too. To assess if the total system is in a healthy state (can cover its obligations), one needs to calculate the intended values of the different categories and see if the sum matches the amount of tokens available on-chain. This is complicated and hence not easily accessible for the end user.

Especially assessing the due rewards (or the claimable rewards) is not trivial as this accounting is partially done at the individual ValidatorShare contract. This has to be evaluated for each delegator of each validator individually.

An existing bug in the rewards distribution was not caught earlier, which highlights the need for monitoring of the systems operation / status.

7.2 `Validator.status` Is Unintuitive

Note Version 1

`Validator.status` is an enum with 4 different possible states.

```
enum Status {Inactive, Active, Locked, Unstaked}
```

When a validator is part of the active validator set and allowed to participate in consensus by signing checkpoints, it has the Status `Active`. When it is unstaked, `_unstake()` does not change the Status. It will remain `Active` even though the validator will no longer be allowed to sign checkpoints and its stake is no longer counted towards the total stake in the system. The Status is only changed to `Unstaked` once `unstakeCLaim()` is called. This is only possible after at least waiting for `WITHDRAWAL_DELAY` to pass. However, it could also be called much later or never.

An external observer trying to understand the system state will likely assume that all validators with Status `Active` are participating in consensus, but this is not the case.