PUBLIC

Code Assessment

of the POL Transition

Smart Contracts

August 28, 2024

Produced for

polygon

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Polygon team,

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 POL Transition according to Scope to support you in forming an opinion on their security risks.

Polygon implements several changes to the Polygon ecosystem that consolidate the transition to the POL token, the native token for Polygon 2.0 and the successor to the MATIC token.

The most critical subjects covered in our audit are correctness of the proxy upgrade and the overall functional correctness. Security regarding both subjects is high after Storage Collisions have been mitigated.

The documentation of the codebase is improvable.

In summary, we find that the codebase provides a high 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	1
• Code Corrected	1
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2
• Code Corrected	2

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 POL Transition repository based on the documentation files.

This report analyzes the impact of three different Pull Requests (PRs) on the following contracts. In particular, this security review was performed only over the changes introduced by the PRs, assuming the underlying codebase (without the PRs) to be bug-free. No other contracts have been examined.

- **pos-contracts**:
    - IStakeManager.sol
    - StakeManager.sol
    - StakeManagerStorage.sol
    - ValidatorShare.sol
- **pol-token**:
    - IDefaultEmissionManager.sol
    - DefaultEmissionManager.sol

The table below indicates the code versions relevant to this report and when they were received.

pos-contracts (PR 4)

V	Date	Commit Hash	Note
1	03 June 2024	eafb462c3b0ae5b63408dcff15b028ee07c0b606	Initial Version
2	24 July 2024	f6cc0d864dc1f7dacad378301b9c527c10fa03c7	After Intermediate Report
3	25 July 2024	39abd8808bcf404726593c3fe10f2bfd4a6e6fea	After Bug Report

pol-token (PR 58)

V	Date	Commit Hash	Note
1	03 June 2024	ea925aa11c5243a7f49b8f67c78d60e54a399622	Initial Version
2	07 June 2024	91d4f8fbe8a63a9c647d289e4116e34c96fc835f	After Intermediate Report

pol-token (PR 60)

V	Date	Commit Hash	Note
1	03 June 2024	8b0a0b6a7c9a41e0a4ed8fc7961c10d6a5b9cefb	Initial Version

For the solidity smart contracts, the compiler version `0.5.17` (pos-contracts) and `0.8.21` (pol-token) was chosen.

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section. In particular, tests, scripts, external dependencies, and configuration files are not part of the audit scope.

2.2 System Overview

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

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Polygon implements several changes to the Polygon PoS ecosystem that consolidate the transition to the POL token, the native token for Polygon 2.0 and the successor to the MATIC token. To achieve the transition, the POL token has been set as the default token for all functionalities in the staking contract. In addition, the emission rates of POL have been adjusted to conform with the current MATIC reward schedule.

Overall, the audit was carried out on the differences introduced by the following Pull Requests (PR):

- pos-contracts/PR4: Sets POL as the default token for all operations.
- pol-token/PR58: Lowers the emission rate of POL according to PIP-26 (Archived version as of June 07 2024: Link).
- pol-token/PR60: Sends the emissions to the StakeManager in POL instead of MATIC.

Below, we give an overview of the contracts involved in the mentioned modifications.

2.2.1 StakeManager

Polygon's Proof of Stake network (Polygon PoS) works with smart contracts on Ethereum for staking management. Stakers of POL tokens (Validators) sign on checkpoints of Polygon PoS. The RootChain contract submits checkpoints and signatures to the StakeManager contract for verification. 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 POL 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 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 POL tokens. **These POL tokens need to be provided to the contract. The staking contract holds a significant amount of POL tokens (most being stake of the Validators). It cannot distinguish between POL tokens belonging to stake or reward. If no or insufficient additional POL tokens are provided to the contract, reward may be paid out using the staked tokens of the validators.** Additionally, the StakeManager holds the POL 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 105.

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.

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.

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.

2.2.2 PolygonEcosystemToken

This contract defines the POL ERC20 token which has built-in integrations for EIP-2612 and Permit2. The initial supply is 10 billion tokens to match the MATIC supply, but more tokens can be minted through emissions.

The contract uses role-based access control for permissioned operations. An address with the default administrator role can grant and remove roles to other addresses. Custom roles are:

- `EMISSION_ROLE`: Can mint a limited amount of tokens per unit of time. It is expected to be the `DefaultEmissionManager`.
- `CAP_MANAGER_ROLE`: Can set the emissions limit. It is expected to be the governance.
- `PERMIT2_REVOKER_ROLE`: Can control the universal allowance granted to Permit2.

The `DEFAULT_ADMIN_ROLE` is expected to be the governance.

The emissions are bounded by the `mintPerSecondCap` parameter. It is set to 13.37 POL per second by default and can be updated arbitrarily by an address granted the `CAP_MANAGER_ROLE`. When `mint()` is called, the number of tokens that can be minted is capped by an amount relative to the time delta since the last mint event, i.e., `(block.timestamp - lastMint) * mintPerSecondCap`.

If `permit2Enabled` is set to true (the default), the Permit2 contract has unlimited allowance from any account. If it is disabled, then the default behavior applies, and individual accounts can choose to set an allowance for the Permit2 contract.

2.2.3 DefaultEmissionManager

This contract defines the fine-grained emission policy for the POL token. It allows the minting of 2.5% of the total supply of POL, compounding. 1.5% of the minted amount is distributed to the stake manager and the remaining 1% to the treasury contracts. The stake manager receives the emission in the form of POL tokens.

The unmigration can be done only if there is enough MATIC token in the migration contract and if the unmigration is unlocked. The first case is assumed to be met by Polygon that specified:

> we anticipate Polygon ecosystem participants will be migrating MATIC to POL in order to provide sufficient one-way liquidity on PolygonMigration.sol

2.2.4 PolygonMigration

This contract allows users to exchange MATIC for POL and vice versa at a 1-to-1 rate. The contract is ownable, and the owner can enable and disable the POL-to-MATIC conversion (unmigration) at will. The owner can also burn POL tokens by sending them to the dead address, so the total supply is not impacted.

The migration contract is assumed to be initialized during the proxy deployment.

2.2.5 Changes in the reviewed PRs

Currently, StakeManager and ValidatorShare expose several functions that can be called by validators and delegators to transfer MATIC tokens between themselves and the contracts. The new revision of the contracts changes this in favor of the POL token. Since POL is a 1:1 translation of MATIC, the functionality does not change and variables containing MATIC amounts are now re-purposed for POL amounts.

To allow for a smooth transition, for most of the functions an equivalent function has been added that still allows to transact with MATIC. These are typically suffixed by the term `Legacy`. For example, `stakeFor()` now accepts POL tokens and the new function `stakeForLegacy()` exposes the same functionality but accepts MATIC tokens.

The POL token's DefaultEmissionsManager emits less new POL tokens to the StakeManager as the token emissions are being reduced according to the plan in the respective PIP-26 (Archived version as of June 07 2024: Link).

2.2.6 Changes in Version 2 of the POS contracts

`StakeManager` and `ValidatorShare` no longer contain the `Legacy` suffix in the names of functions transacting with MATIC. Instead, functions transacting with POL now contain the suffix `POL` in their name. This is, however, not the case for functions that do not have two versions: `dethroneAndStake()`, for example, handles POL tokens even though it does not contain the `POL` suffix.

Additionally, the functionality of `StakeManager.slash()` has been removed.

2.2.7 Trust Model

Users of the system are generally untrusted and expected to behave unpredictably.

The following roles are fully trusted and expected to behave honestly and correctly.

- The administrator, the cap manager, and the Permit2 revoker of the POL token contract.
- The owner of the migration contract.
- The Governance in the StakeManager.
- The owner of the proxy contracts.

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.

The addresses granted the pauser role in the MATIC contract are also relevant. Migration/unmigration can be indirectly paused by pausing the MATIC token.

Moreover, the `EMISSION_ROLE` of the `PolygonEcosystemToken` is assumed to be granted to the `DefaultEmissionManager`.

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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

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	1

- Storage Collisions Code Corrected

High-Severity Findings	0

Medium-Severity Findings	0

Low-Severity Findings	2

- Fee Claiming in MATIC Only Code Corrected
- Unused Functions Code Corrected

Informational Findings	3

- Unreachable Initializer Code Corrected
- Typographical Errors Code Corrected
- Missing Gap Change Code Corrected

6.1 Storage Collisions

Security Critical Version 1 Code Corrected

CS-POLY_TO_POL-001

The StakeManager contract is deployed behind the upgradeable proxy StakeManagerProxy. StakeManager inherits from the following contracts:

- StakeManagerStorage
- Initializable
- IStakeManager
- DelegateProxyForwarder
- StakeManagerStorageExtension

Therefore, the storage of the proxy is laid out according to the storage variables in the above contracts. It is worth mentioning that the Initializable contract holds the `inited` variable that prevents the StakeManager from being reinitialized.

During the changes under review, two new storage variables were added to the StakeManagerStorage contract: `tokenLegacy` and `migration`.

The addition of these variables in StakeManagerStorage creates a storage collision in StakeManagerProxy. In particular, after the upgrade, `inited` will be read from the slot in which `rewardPerStake` is stored. Due to the new storage layout, `inited` and `migration` are now read from the same slot, using an offset of 160 bytes for `inited`. As long as the number in `rewardPerStake` is less than 160 bytes long (which it currently is on-chain), `inited` will be read as 0.

The consequences of this are dire: right after the upgrade, anyone can reinitialize the contract, making themselves (or any arbitrary address) both owner and governor of the StakeManager. After that, this adversary could perform any privileged function of the StakeManager. This includes, but it is not limited to, completely draining the StakeManager of all its POL.

Code corrected:

The new storage variables have been moved to `StakeManagerStorageExtension`, resolving the storage collision.

6.2 Fee Claiming in MATIC Only

Design | Low | Version 2 | Code Corrected

CS-POLY_TO_POL-009

`StakeManager.claimFee()` transacts using MATIC. There is, however, no alternative that uses POL tokens.

Code corrected:

`claimFee()` now transacts using POL.

6.3 Unused Functions

Design | Low | Version 1 | Code Corrected

CS-POLY_TO_POL-002

Some new *legacy* functions in the StakeManager are never actually called:

1. `dethroneAndStakeLegacy()` can only be called from the StakeManager itself. There is, however, no code that actually performs such a call. `dethroneAndStake()` is called from `confirmAuctionBid()` in the StakeManagerExtension but the function offers no alternative legacy version.
2. `migrateOutLegacy()` and `migrateInLegacy()` in ValidatorShare are never called from the StakeManager (the only contract allowed to call the functions). StakeManager exposes a function `migrateDelegation()` that calls `migrateout()` and `migrateIn()` but offers no alternative legacy version.

Code corrected:

The mentioned functions have been removed.

6.4 Missing Gap Change

Informational | Version 1 | Code Corrected

CS-POLY_TO_POL-005

The new storage variable `START_SUPPLY_1_2_0` is added to DefaultEmissionManager. The contract defines a `_gap` at the end of its storage layout that is supposed to keep the storage layout for inheriting contracts in order. Since a new variable is added before the gap, the gap should be reduced by 1.

Because no other contracts inherit from DefaultEmissionManager, this missing change does not represent a problem.

Code corrected:

The `_gap` has been updated correctly.

6.5 Typographical Errors

Informational **Version 1** **Code Corrected**

CS-POLY_TO_POL-007

The following is a non-exhaustive list of typographical errors in the code:

1. `StakeManager._unstake()` contains a comment stating "*if validators unstake and slashed to 0, he will be forced to unstake again*".
2. `ValidatorShare` contains a comment stating "*all matic will be staken in one go*".
3. `ValidatorShare.__sellVoucher()` contains a comment stating "*undond period*".

Code corrected:

Errors #2 and #3 have been fixed.

6.6 Unreachable Initializer

Informational **Version 1** **Code Corrected**

CS-POLY_TO_POL-008

StakeManager adds two new storage variables `tokenLegacy` and `migration`. The variables are initialized in the function `initialize()` of the contract. Since the contract is already live behind a proxy, and the `initialize()` function has already been called on the proxy, the function can not be called again after an implementation update as it is only callable once (i.e., the related `inited` variable must be `false` to be able to call the function but it is currently set to `true`).

The variables can, however, still be set via their respective setter functions.

Code corrected:

The variables can now be set with the new initializer `initializePOL()`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Inconsistent Naming

Informational **Version 1**

CS-POLY_TO_POL-003

The function name `updateTimeline()` in the StakeManager is `internal` but does not start with an underscore. This is not consistent with the naming of other functions in the contract.

7.2 Missing Events

Informational **Version 1**

CS-POLY_TO_POL-004

Some state-changing functions in StakeManager do not emit events. This is particularly true for most setter functions such as `setLegacyToken()` or `setMigration()`.

7.3 Missing NatSpec / Code Documentation

Informational **Version 1**

CS-POLY_TO_POL-006

Neither StakeManager nor ValidatorShare provide any code documentation besides notes that simply state the function name (in some cases).

8 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.

8.1 Ambiguous Naming

Note Version 2

The contract upgrade introduces several new functions that allow users to interact with the contracts using POL tokens instead of MATIC tokens. These new functions' names are composed of the name of the function that transacts using MATIC and the suffix `POL`.

However, some functions (e.g., `StakeManager.topUpForFee()`) don't have a second version and are transacting with POL while their name does not contain the `POL` suffix.