PUBLIC

Code Assessment

of the ClayStack Matic Smart Contracts

March 23, 2022

Produced for

ClayStack

by

CHAINSECURITY

Contents

1 Executive Summary

Dear ClayStack team,

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

ClayStack implements a staking pool implementation that simplifies the staking MATIC tokens on numerous Polygon validators. A user that joins the pool, locks MATIC tokens and gets csMATIC tokens that accumulate the rewards over time. The csMATIC tokens can be then burned, to get the locked MATIC tokens back.

The most critical subjects covered in our audit are the security of the pool and token contracts, the functional correctness and the safety of the deposited funds. Security regarding all the aforementioned subjects is high.

In the final iteration after the intermediate reports no issues remain open. Overall we find that the codebase in its current state 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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	12
• Code Corrected	11
• Specification Changed	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 following source code files inside the ClayStack Matic repository based on the documentation files.

- ClayMatic.sol
- IClayMain.sol
- CsToken.sol
- RoleManager.sol

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

V	Date	Commit Hash	Note
1	31 Jan 2022	74fe4e52b696c53fd5e0fbbf858fc2f06086f232	Initial Version
2	11 Mar 2022	ebd2d78e38af34c9eb4cca4bfb9745126563a850	Version with fixes

For the solidity smart contracts, the compiler version `0.8.11` was chosen.

2.1.1 *Excluded from scope*

All source files inside the ClayStack Matic repository, that are not mentioned in Scope, are not part of this assessment. The imported libraries and contracts are excluded from the scope as well.

2.1.2 *Deployment*

The deployment was verified based on a modified version of the code utilizing OpenZeppelin 4.5.0, while the audit was based on OpenZeppelin 4.4.2.

We validated deployment at the following addresses on Ethereum Mainnet for block `14396490`:

Contract	**Ethereum Mainnet address**
ClayMatic: Proxy	`0x91730940DCE63a7C0501cEDfc31D9C28bcF5F905`
ClayMatic: Implementation	`0xD634626DE5C6237CE3eaa6805B102Bb8CE9a0e9E`
csToken	`0x38b7bf4eecf3eb530b1529c9401fc37d2a71a912`

Deployment verification ensures that the code we audited has been deployed at the specified addresses. In cases where upgradeable contracts are used, we also validate that the proxy uses the correct implementation contract at the specified block.

We do not validate any state parameters set during deployment or afterwards and users should take caution to ensure that the contract is configured correctly and that any proxies utilize verified and trusted implementation contracts.

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.

ClayStack offers a central pool for clients to stake their MATIC tokens across many different validators. Users can deposit tokens into the pool and later withdraw them. The system itself then stakes the deposited tokens in the validators. It can be configured to distribute the tokens in varying proportions between the different validators. Some tokens are left in the contract so they can be instantly withdrawn. Users can also withdraw normally, in which case an order is placed at one or more validators and are subject to their unbonding period. Once this period is over, the user can claim their tokens to receive them.

The rewards are not calculated at every operation, instead the `autobalance` function should be called at regular intervals. ClayStack states they will do so regularly, but users could also do so themselves. There are different fees for deposits, withdrawals, instant withdrawals, and a portion of the staking rewards are kept for the system as well.

The system is administrated by a timelock contract which can make changes such as adding validator nodes, changing the distribution of staked tokens across the validators, setting the proportion of tokens which should be kept liquid for instant withdrawals, setting the various fees, and setting the vault which collects the fees.

The contract is also upgradable through a timelock.

Smaller, less impactful changes can be made by users with the `CS_SERVICE_ROLE` permissions. They can migrate staked funds between validators, deactivate nodes without any staked funds, set a deposit limit, set a change limit for the exchange rate, set a maximum percentage which can be withdrawn from a single node, set a maximum number of nodes which can be withdrawn from, enable or disable slashing protection, set the over-staking threshold, and pause or unpause the contract. Note that in its current state, pausing the contract prevents any funds from being withdrawn.

The staking happens in a round-robin style, based on a points system to decide which node gets what proportion of the staked funds. Each node is assigned a points value, and it should receive funds proportional to its points value relative to the total number of points. In the round robin system, each node in turn receives an amount of funds which brings them to the correct proportion. This means if a node already has too many tokens, it will not receive any, or if it has too few, it could receive all of them. If there would only be a small amount (defined by the over-staking threshold) of tokens left over for the next node, it will be added to the current one instead. Thus, in the long term the system should converge to a distribution of staked tokens proportional to the points of each node. The next staking round robin will start with the last node that received tokens.

The unstaking occurs in a similar fashion, but in reverse order. The round-robin goes through each node in turn and unstakes tokens up to a limit, defined by `maxWithdrawNodePercentage`. However, one can't withdraw from more than `maxNodesToWithdraw` different nodes. The unstaked tokens can't be withdrawn until the unbonding period of the validators is up. The order IDs of the unbonding orders are given to the user, which they can use to claim the tokens once they can be withdrawn.

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
- 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	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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	12

- Incorrect Permissions Specification Changed
- Inefficient Modifier Code Corrected
- Logic Contract Code Corrected
- Missing Check Code Corrected
- Missing whenNotPaused in Migrate Delegation Code Corrected
- No Limit On Withdrawal and Deposit Fee Amounts Code Corrected
- Possible Underflow Code Corrected
- Reuse balanceOf Result When Possible Code Corrected
- Staking Issues Code Corrected
- Storage Operations in Loops Code Corrected
- Validator Contract Check Not Strict Enough Code Corrected
- Wrong Variable Logged Code Corrected

6.1 Incorrect Permissions

Correctness Low Version 1 Specification Changed

The `setDefaultLiquidity` function has a doc comment which states it should only be callable with the `TIMELOCK_ROLE`. However, the implementation checks if the caller has the `CS_SERVICE_ROLE`.

```
/**
 * ...
 * @notice Only `TIMELOCK_ROLE` callable.
 * ...
 */
function setDefaultLiquidity(uint256 value) external onlyRole(CS_SERVICE_ROLE) {
    require(value < PERCENTAGE_BASE, "CM006");
    defaultLiquidity = value;
}
```

Specification Changed:

The doc comment was changed to specify the function to be only callable with the `CS_SERVICE_ROLE`.

6.2 Inefficient Modifier

Design Low Version 1 Code Corrected

The CsToken contract makes use of the `onlyClayMain` modifier, which needs to know what the address of the ClayMatic contract is. Instead of implementing a storage value that can only be set once, an immutable variable could be used. This would also be far more gas-efficient, since immutable variables do not incur storage reads.

Code corrected:

The storage variable was made immutable and the `onlyOnce` modifier removed.

6.3 Logic Contract

Security Low Version 1 Code Corrected

The `ClayMatic` and `RoleManager` contracts have problems with their UUPS logic contracts:

1. The `initialize` function is unprotected on the logic contracts.
2. The `upgradeTo` function overrides the `UUPSUpgradeable` function, but does not use the `onlyProxy` modifier.

Since there are no unprotected delegateCalls available, the effect of these problems is limited. But one can set the storage variables of logic contracts to any values. Consider using `onlyProxy` for functions that shouldn't be called on logic contracts directly.

Code corrected:

The `onlyProxy` modifier was added to the `initialize` and `upgradeTo` functions.

6.4 Missing Check

Correctness Low Version 1 Code Corrected

The comment on the function `setMaxWithdrawNodePercentage` states:

```
/**
 * ...
 * Requirements:
 * - `value` can not be zero.
 * ...
 */
function setMaxWithdrawNodePercentage(uint256 value) external onlyRole(CS_SERVICE_ROLE) {
    require(value <= PERCENTAGE_BASE, "CM006");
    maxWithdrawNodePercentage = value;
}
```

However, there is no check to make sure `value` is not equal to zero.

Code corrected:

A check was added to make sure `value` is not equal to zero.

6.5 Missing `whenNotPaused` in Migrate Delegation

Design Low Version 1 Code Corrected

All balance affecting functions have the `whenNotPaused` modifier applied except for `migrateDelegation`, this seems like an oversight if pause is meant to be used in emergency or upgrade situations where critical contract state should not change in between upgrades.

Code corrected:

The `whenNotPaused` modifier was added to the `migrateDelegation` function.

6.6 No Limit On Withdrawal and Deposit Fee Amounts

Design Low Version 1 Code Corrected

There is currently no limit on any fee amounts besides being below 100%, but this should not be the case from both a trust and correctness perspective. The holder of `TIMELOCK_ROLE` could set instant or regular withdrawal fees to 100% to prevent anyone from withdrawing, or increase deposit fee to 100% to basically prevent anyone from staking any more funds without losing them all, effectively disabling those functions in a round-about way.

Consider setting hard limits in the contract beyond which fees cannot be raised without a total contract upgrade.

Code corrected:

Maximum values were added to all fee types.

6.7 Possible Underflow

Correctness Low Version 1 Code Corrected

In the function `_balanced` the following check is done:

```
uint256 stakingFlow = (underlyingToken.balanceOf(address(this)) + funds.stakedDeposit) / 1e10;
require(stakingFlow - 1e6 <= userFlow && userFlow <= stakingFlow + 1e6, "CMB01");
```

However, if `underlyingToken.balanceOf(address(this)) + funds.stakedDeposit` is less than `1e16`, this will result in an underflow and revert, despite not necessarily being an invalid state.

Code corrected:

The code was refactored so the underflow is no longer possible.

6.8 Reuse `balanceOf` Result When Possible

Design | Low | Version 1 | Code Corrected

There are various functions which call `underlyingToken.balanceOf(address(this))` multiple times. While in some cases, the balance does change and the additional cross-contract call is necessary, in others it is not. Therefore, the redundant calls could be omitted to save gas.

1. In `autoBalance` and `_balanced`, the balance of the contract is queried twice without any balance change in between.
2. In `_sequentiallyStake`, the balance of the contract is queried once per loop iteration. While the balance can change between iterations, it may instead be possible to check that the balance is greater than the total amount to stake, rather than checking individually for each staking operation.

Code corrected:

1. The mentioned functions were updated to query the balance only once.
2. The total amount to stake is now compared to the balance at the start of the function, instead of once per loop iteration.

6.9 Staking Issues

Design | Low | Version 1 | Code Corrected

When autobalance is run, if the additional amount to stake is consistently smaller than the `overStakingThreshold`, the same validator will be chosen every time. This is because the `activeStakingNode` does not change if only the first node is used. Thus, the amounts staked per validator will not converge if the amounts to stake per balancing operation are consistently below `overStakingThreshold`.

Code corrected:

The `activeStakingNode` is now advanced by one position at the end of the function to avoid the mentioned issue.

6.10 Storage Operations in Loops

Design | Low | Version 1 | Code Corrected

Many different functions contain loops which access storage variables. Rather than reading a storage variable once per loop iteration, it is more gas-efficient to read it once before the loop and cache the value in a local variable.

There are also loops which write to storage variables. Again, rather than writing to storage directly in the loop, it's more gas-efficient to write to a local variable and move the final value to storage after the loop.

These optimizations can be applied in the following functions:

1. In the function `claim`, the value `withdrawOrders[msg.sender]` can be stored in a variable outside the loop and reused.

2. In `getMaxWithdrawAmountCs`, the storage values `maxNodesToWithdraw` and `maxWithdrawNodePercentage` can be cached.
3. In `_sequentiallyStake`, the number of storage writes to `activeStakingNode` and `funds.stakedDeposit` can be reduced significantly. Additionally, the values `totalPoints`, `overStakingThreshold`, `stakeManager` and `underlyingToken` could be cached. Note that it may not be possible to apply all these optimizations without causing a "Stack too deep" compilation error.
4. In `_sequentiallyUnstake`, the number of writes to `activeUnstakingNode` can be reduced. The values of `maxNodesToWithdraw` and `maxWithdrawNodePercentage` can be cached to reduce storage reading operations.
5. In `_getMaxWithdrawAmount`, the values of `maxWithdrawNodePercentage` and `maxNodesToWithdraw` can be cached.
6. In `addNodes`, the number of writes to `totalPoints` can be reduced by calculating the value in a local variable and writing to storage after the loop. Similarly, a local variable could be introduced to hold the value of `countStakingNodes` and the final value written back only at the end. Lastly, `stakeManager` could be cached so it only has to be read once before the loop.
7. In `autoBalance`, `_updatedStaked`, `_isNodeActive` and `_updateNodePoints`, the value of `activeNodes.length` can be stored locally to reduce storage reads.

Code corrected:

The suggested changes were made. In the case of `_sequentiallyUnstake`, `maxNodesToWithdraw` was not cached due to the "Stack too deep" compilation error.

6.11 Validator Contract Check Not Strict Enough

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

Inside of *ClayMatic.addNodes*, the validity of a validator is checked in the following way:

```
require(stakeManager.isValidator(val.validatorId()) && !val.locked(), "CM010");
```

Both values validated against are sourced from the supplied contract which could just be a malicious contract lying about being a validator. A check with stronger correctness would be to require that the following is true:

```
stakeManager.getValidatorContract(validatorId) == val
```

This would prevent adding an invalid validator contract by mistake.

Code corrected:

The suggested check was implemented.

6.12 Wrong Variable Logged

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

In the `autoBalance` function, any accrued fees are transferred to the vault. Additionally, an event is emitted to log the transferred fees. However, the wrong variable is used for the event so the emitted value will always be zero.

```
if (funds.accruedFees != 0) {
    uint256 accruedFees = funds.accruedFees;
    funds.accruedFees = 0;
    underlyingToken.safeTransfer(vaultManager, accruedFees);
    emit LogTransferVault(vaultManager, funds.accruedFees);
}
```

Code corrected:

The correct variable is now used to log the event.

6.13 Consider Using Modifiers for *_Balanced* and *_updateBalance* Functionality

Note | Version 1 | Code Corrected

The calls to *_balanced* and *_updateBalance* could be more cleanly implemented via a modifier like the following:

```
modifier enforceAndUpdateBalance {
 _updateBalance();
 _;
 _balanced();
}
```

It would prevent needing to manually ensure both are called, in the right order and priority, in any future update to the code and simply enforce the presence of this modifier on the relevant functions.

Code corrected:

The suggested modifier was introduced and applied to all relevant functions.

6.14 Unnecessary Require

Note | Version 1 | Code Corrected

In the `claim` function, the following require statement is unnecessary:

```
require(amountAvailable >= userAmount + payableFee, "CMC02");
```

The value of `userAmount` is essentially calculated as `receivedAmount - payableFee`. The current balance (`amountAvailable`) cannot be smaller than the amount received, therefore this check will never fail.

Code corrected:

The unnecessary require statement was removed.