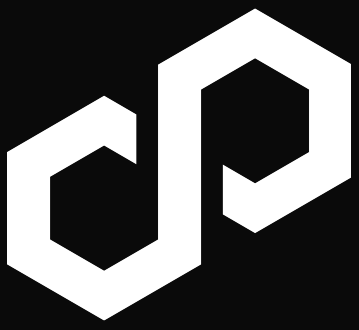
Security Assessment
Final Report



Kat token

March 2025

Prepared for Polygon Labs

certora

Table of Contents

Project Summary

Project Scope

Project Name	Repository (link)	Commit Hash	Platform
Kat Token	https://github.com/0xPolygon/kat-token/tree/main	21644ff	EVM

Project Overview

This document describes the specification and verification of **Kat Token** using the Certora Prover and manual code review findings. The work was undertaken from **11.3.2025** to **18.3.2025**.

The following contract list is included in our scope:

```
kat-token/src/KatToken.sol
kat-token/src/MerkleMinter.sol
kat-token/src/Powutil.sol
```

The Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

All issues identified in the audit were resolved as of commit eea36ab.

Protocol Overview

The protocol implements Kat Token, an ERC20 which is the native token of the Katana network chain. An initial amount of tokens could be minted and distributed by the Merkle Minter, a smart contract that mints tokens for valid leaves in a Merkle Tree. After 4 years, a limited amount of token inflation starts, allowing the fee beneficiary to mint new Kat Tokens.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	3	3	3
Informational	8	7	6
Gas Optimization	4	4	1
Total	15	14	10

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
L-01	Renounced roles in KatToken can be re-acquired	Low	Fixed
L-02	MerkleMinter's root and katToken can be changed after claims have happened	Low	Fixed
I-01	Off-by-one error in MAX_INFLATION check	Info	Fixed
I-02	Deployment logic could be simplified	Info	Fixed
I-03	KatToken.renounceInflationBeneficiary() allows distributing minting capacity to address(0)	Info	Fixed
I-04	MerkleMinter.rootSetter is allowed to renounce before calling MerkleMinter.init	Info	Fixed
I-05	18-digits constants are saved with 17 digits of precision	Info	Fixed
G-01	Unchecked subtraction in KatToken.distributeMintCapacity	Gas	Acknowledged

G-02	KatToken.distributeInflation could return early	Gas	Fixed
G-03	KatToken._calcInflation could return early	Gas	Acknowledged
G-04	MerkleMinter.claimKatToken spends unnecessary gas in double hashing	Gas	Acknowledged

Low Severity Issues

L-01 Renounced roles in KatToken can be re-acquired

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: KatToken.sol	Status: Fixed in 11d9bb7	Violated Property: P-03

Description: The code in scope allows for a few privileged addresses responsible for the initial operation and setup of the contracts. These roles can then be renounced to make the contracts eventually become fully decentralized. This full decentralization can however be reverted because renouncing a role can be undone through a pending role assignment:

```
JavaScript
File: KatToken.sol
82:     function acceptInflationAdmin() external {
83:         require(msg.sender == pendingInflationAdmin, "Not new role owner.");
84:         inflationAdmin = pendingInflationAdmin;
85:         pendingInflationAdmin = address(0);
86:         emit InflationAdminChanged(inflationAdmin, false);
87:     }
---
93:     function renounceInflationAdmin() external {
94:         require(msg.sender == inflationAdmin, "Not role owner.");
95:         inflationAdmin = address(0);
96:         emit InflationAdminChanged(address(0), false);
97:     }
```

Recommendations: Clear also pending addresses when a `katToken` role (`inflationAdmin`, `inflationBeneficiary`) is renounced. This would be in analogy with OpenZeppelin's `Ownable2Step` behavior when `renounceOwnership()` is called.

Customer's response: Fixed in 11d9bb7.

Fix Review: Issue Fixed.

L-02 MerkleMinter's root and katToken can be changed after claims have happened

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: MerkleMinter.sol	Status: Fixed in c19f230	Violated Property: P-02

Description: The `MerkleMinter` contract allows the `rootSetter` address to call its `init` function multiple times, effectively allowing changing the airdrop information and its ERC-20 `katToken` until `rootSetter` renounces its role.

It is however possible that `rootSetter` updates the Merkle tree `root` and/or the ERC-20 `katToken` ***after*** some users have successfully claimed their share of tokens.

Exploit Scenario: When this happens, the airdrop can end up in an inconsistent state because:

- Parts of the airdrop can happen and distribute different ERC-20s in case `katToken` changed
- Depending on how the Merkle tree is computed off-chain, it is possible that equal airdrop bits (`receiver`, `amount` pairs) are assigned a different `index`, thus allowing a single `receiver` to claim the same amount twice, while another `receiver` can't ever claim their share

In the case of a malicious `rootSetter`, this can also be abused to selectively deny users with legitimate leaves from minting KatTokens even when there's an appearance that everything was configured properly.
In more detail, the `rootSetter` can initially set the expected root with an incorrect token as the `katToken`. They will then call `claimKatToken()` at some users' behalf, setting their leaves as claimed, then calling `init()` once again with the correct root and the correct `katToken` followed by a call to `renounceRootSetter()`, providing the appearance that all users with a valid leaf would be able to claim their Kat Tokens.

Recommendations: Forbid calling `init()` as soon as `claimKatToken()` becomes callable.

Customer's response: Fixed in c19f230.

Fix Review: Issue Fixed. Note that the added check is not completely inverted to the one in `claimKatToken()` because it does not use `>=`.

```
JavaScript
File: MerkleMinter.sol
42:         require(((block.timestamp < unlockTime) && locked), "Minter not locked.");
---
82:         require(((block.timestamp > unlockTime) || !locked), "Minter locked.");
```

Informational Issues

I-01. Off-by-one error in MAX_INFLATION check

Description: The `katToken`'s `changeInflation()` function accepts values up to `MAX_INFLATION`:

```
JavaScript
File: KatToken.sol
156:     function changeInflation(uint256 value) external {
157:         require(msg.sender == inflationAdmin, "Not role owner.");
158:         require(value < MAX_INFLATION, "Inflation too large.");
```

This check, which leaves `MAX_INFLATION` off the range of accepted values, may provide a bad user experience: calling `changeInflation(MAX_INFLATION)` would fail, despite `MAX_INFLATION` would more intuitively be seen as the maximum attainable and not the minimum unattainable value.

Recommendation: Consider updating the check to be "`require(value <= MAX_INFLATION, …)`".

Customer's response: Fixed in 600a24f.

Fix Review: Issue Fixed.

I-02. Deployment logic could be simplified

Description: The deployment logic in `Deploy.s.sol` contains some steps to pre-compute contract addresses to solve circular dependencies:

```
JavaScript
File: Deploy.s.sol
27:     function deploy(
---
33:     ) public returns (KatToken katToken, MerkleMinter merkleMinter) {
34:         address _merkleMinter = vm.computeCreate2Address(
35:             salt,
36:             keccak256(bytes.concat(type(MerkleMinter).creationCode,
abi.encode(_unlockDelay, _unlocker, _rootSetter)))
37:         );
---
43:         katToken = new KatToken{salt: salt}("KatToken", "KAT", _inflation_admin,
_inflation_ben, _merkleMinter);
---
47:         merkleMinter = new MerkleMinter{salt: salt}(_unlockDelay, _unlocker,
_rootSetter);
48:
49:         assert(address(merkleMinter) == _merkleMinter);
50:     }
```

This complexity is however unnecessary because the dependency is one-way before `MerkleMinter.init()` is called - `katToken` needs `MerkleMinter`'s address, but `MerkleMinter` doesn't need `KatToken`'s address at construction.

Recommendation: Consider simplifying the deployment by:

1. Normally deploying `MerkleMinter` first.
2. Deploying `katToken`, with a reference to `MerkleMinter`.
3. (optionally) calling `MerkleMinter.init()` with `katToken`'s address.

Customer's response: Fixed in 8631f84.

Fix Review: Issue Fixed.

I-03. KatToken.renounceInflationBeneficiary() allows distributing minting capacity to address(0)

Description: Whenever `KatToken.renounceInflationBeneficiary()` is called, it is possible that `inflationFactor` is non-zero. When this happens, new inflation will be minted to the zero address, and this is likely an undesirable situation.

Recommendation: Consider enforcing `inflationFactor` to be zero when `inflationBeneficiary` is `address(0)`.

Customer's response: Fixed in 7838454.

Fix Review: Issue Fixed.

I-04. MerkleMinter.rootSetter is allowed to renounce before calling MerkleMinter.init

Description: The function `MerkleMinter.renounceRootSetter()` allows the `rootSetter` to give up its role after it has properly configured the contract via the `init()` function call. There is however no check that the `init()` call actually happened beforehand.

Recommendation: Consider adding a `require(root != bytes32(0))` check in `renounceRootSetter`.

Customer's response: Fixed in b5bdc1b.

Fix Review: Issue Fixed.

I-05. 18-digits constants are saved with 17 digits of precision

Description: The `inflationFactor` constants are saved with 17 digits of precision, effectively using 0 as the least significant digit which could have been used to represent a meaningful digit. Ultimately, this affects the precision of the output of the `exp2()` function:

```
JavaScript
exp2(42644337408493690) = 1030000000000000003
exp2(42644337408493685) = 1029999999999999999
---
exp2(28569152196770890) = 1019999999999999997
exp2(28569152196770894) = 1020000000000000000
```

Recommendation: Correct the constants to account for the full possible precision.

```
JavaScript
File: KatToken.sol
29:         uint256 public constant MAX_INFLATION = 0.042644337408493685e18; // log2(1.03).
Currently 0.04264433740849369e18
---
63:             inflationFactor = 0.028569152196770894e18; // log2(1.02). Currently
0.02856915219677089e18
```

Customer's response: Fixed in aa2932f.

Fix Review: Issue Fixed.

Gas Optimization

G-01. Unchecked subtraction in KatToken.distributeMintCapacity

Description: The `distributeMintCapacity` function implements a check at `L201` for `amount` to not exceed the sender's capacity. This makes the `L202` implicit underflow check unnecessary.

JavaScript

```
File: KatToken.sol
199:     function distributeMintCapacity(address to, uint256 amount) external {
200:         require(to != address(0), "Sending to 0 address");
201:         require(mintCapacity[msg.sender] >= amount, "Not enough mint capacity.");
202:         mintCapacity[msg.sender] -= amount;
203:         mintCapacity[to] += amount;
204:         emit MintCapacityDistributed(msg.sender, to, amount);
205:     }
```

Recommendation: Consider adding an `unchecked` block around `L202`.

Customer's response: Acknowledged. Won't be fixed because the change reduces readability for a seldom called function.

G-02. KatToken.distributeInflation could return early

Description: The `distributeInflation` function implements a check at `L188` for incrementing or not `lastMintCapacityIncrease`:

```
JavaScript
File: KatToken.sol
182:     function distributeInflation() public {
183:         uint256 inflation = _calcInflation();
184:         distributedSupplyCap += inflation;
185:         // give increase to INFLATION_BENEFICIARY
186:         mintCapacity[inflationBeneficiary] += inflation;
187:         // increase distributedSupplyCap
188:         if (block.timestamp > lastMintCapacityIncrease) {
189:             lastMintCapacityIncrease = block.timestamp;
190:         }
191:         emit InflationDistributed(inflationBeneficiary, inflation);
192:     }
```

However, under the same condition of `block.timestamp > lastMintCapacityIncrease`, it is guaranteed that inflation is zero.
Recommendation: Consider moving the `block.timestamp > lastMintCapacityIncrease` check for an early return at the top of the function.

Customer's response: Fixed in 321a624.

Fix Review: Issue Fixed. Note that the new check is inverted but does use `>=`.

```
JavaScript
File: KatToken.sol
188:         if (lastMintCapacityIncrease > block.timestamp) {
189:             return;
190:         }
```

G-03. KatToken._calcInflation could return early

Description: The `_calcInflation` function implements a check at `L170` for early returning in a case where inflation is known to be zero:

```
JavaScript
File: KatToken.sol
169:     function _calcInflation() internal view returns (uint256) {
170:         if (lastMintCapacityIncrease > block.timestamp) {
171:             return 0;
172:         }
173:         uint256 timeElapsed = block.timestamp - lastMintCapacityIncrease;
174:         uint256 supplyFactor = PowUtil.exp2((inflationFactor * timeElapsed) / 365 days);
// when timeElapsed is 1, we have ~310e6
175:         uint256 newCap = (supplyFactor * distributedSupplyCap) / 1e18;
176:         return newCap - distributedSupplyCap;
177:     }
178:
```

This check could be extended to also cover the case where `lastMintCapacityIncrease == block.timestamp`, which is another condition when inflation is trivially zero.

Recommendation: Consider changing the `L170` check as follows:

```
JavaScript
if (block.timestamp > lastMintCapacityIncrease) {
  // calculate & return
}
return 0;
```

Customer's response: Acknowledged. Won't be fixed because the change reduces readability only to save a bit of gas for a case that wouldn't really ever happen (calling cap() in the same block after distributeInflation).

G-04. MerkleMinter.claimKatToken spends unnecessary gas in double hashing

Description: The Merkle tree leaves in `MerkleMinter` are hashed as follows:

```
JavaScript
File: MerkleMinter.sol
83:         bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(index, receiver,
amount))));
```

This double hashing is unnecessary because the (`index`, `receiver`, `amount`) leaf key encodes to 96 bytes which is a safe length for second preimage attacks.

Recommendation: Consider using a single `keccak256` round like done by Uniswap's MerkleDistributor to reduce the gas footprint of the claiming function.

Customer's response: Acknowledged. Won't be fixed because of development time constraints.

Formal Verification

The formal verification has been successfully completed during the late-stage review, with all proofs conducted on the latest version of the code at commit 494205c. The following results reflect the current state of the implementation and are aligned with the most recent codebase. For reference, the findings from the manual audit are provided in Appendix 1, ensuring full traceability between manual and formal review.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.
No longer applicable	The rule is no longer applicable to the fixed code.

General Assumptions and Simplifications

- Harnessing: We sometimes work with contracts inherited from the original contracts. In these "harnessed" contracts we add additional methods, flags etc. for verification purposes without modifying the original code such that any verification result of the harness applies to the original contracts.
- The verified Solidity contracts are compiled with solcX (with via-ir).
- We're using an approximate version of the `exp2()` function in order to speed up the verification process. The approximation behaves as an uninterpreted function where the following axioms hold:
    a. `forall uint256 x < 100: exp2(x *1e18) == 2^x *1e18`
    b. `x < y => exp2(x) <= exp2(y)`

Formal Verification Properties

KatToken

Module Properties

P-01. Integrity of important methods

Status: Verified

Rule Name	Status	Description	Link to rule report
integrityOfChangeRoleHolder **integrityOfAcceptRole** **integrityOfRenounceInflationBeneficiary** **integrityOfRenounceInflationAdmin** **integrityOfDistributeMintCapacity** **integrityOfRenounceLockExemptionAdmin** **integrityOfUnlockAndRenounceUnlocker**	Verified	*The methods update the storage as expected and can only be called by the corresponding privileged addresses.*	*Report after fix*

P-02. Risk assessment

Status: Verified after fix

Rule Name	Status	Description	Link to rule report
onlyAllowedAddressesCanMoveFunds	Verified	*Balance of any address can only change either via mint, or after the token is unlocked, or when msg.sender is exempt.*	*Report after fix*
onceActivatedItStaysActive	Verified	*Once active, the protocol will not go to an inactive state.*	*Report after fix*

rootCanAlwaysBeSet	No longer applicable Reported issue: I-04	*It should not be be possible to permanently renounce the rootSetter role without actually setting the root first*	*Original report*
cannotInitAfterUnlocked	No longer applicable Reported issue: L-02	*init cannot be called after the minter has already become active*	*Original report*
canOnlyClaimWhenNotLocked	No longer applicable	*All calls to claimKatToken must revert before unlockTime*	*Original report*
cannotClaimTheIndexTwice	No longer applicable	*indexIsClaimed(index) == true then claimKatToken(..,index) will revert.*	*Original report*

P-03. Allowed transitions of storage variables

Status: Verified after fix

Rule Name	Status	Description	Link to rule report
lockedValueChange	Verified	*Locked can only change from true -> false*	*Report after fix*
lastMintCapacityIncreaseNeverDecreases	Verified	*lastMintCapacityIncrease can never decrease*	*Report after fix*
distributedSupplyCapNeverDecreases	Verified	*distributedSupplyCap can never decrease*	*Report after fix*

mintCapacityPlusMintedEqualsDistributedSupplyCap	Verified	*distributedSupplyCap == total mint capacity + mintedTokens*	*Report after fix*
mintCapacityOfZeroIsZero	Verified after fix Reported issue: I-03	*mintCapacity of address(0) is always 0*	*Report* *Report after fix*
changeInflation_revertConditions	Verified after fix Reported issue: I-01	*changeInflation doesn't revert unexpectedly.*	*Report* *Report after fix*
inflationAdminValueChange	Verified after fix Reported issue: I-01	*inflationAdmin can never change from 0 to non-zero*	*Report* *Report after fix*
inflationBeneficiaryValueChange	Verified after fix Reported issue: I-01	*inflationBeneficiary can never change from 0 to non-zero*	*Report* *Report after fix*
noBeneficiaryThenNoInflation	Verified after fix Reported issue: I-03	*If inflationBeneficiary is zero then inflationFactor is zero*	*Report* *Report after fix*
zeroRoleThenZeroPendingRole (covers zeroAdminThenZeroPendingAdmin and zeroBeneficiaryThenZeroPendingBeneficiary)	Verified	*If roleHolder(role) is zero then pendingInRoleHolder(role) is also zero. This implies the following two properties.*	*Report after fix*
zeroAdminThenZeroPendingAdmin	No longer applicable Reported issue: I-01	*If inflationAdmin is zero then pendingInflationAdmin is zero.*	*Original report*
zeroBeneficiaryThenZeroPendingBeneficiary	No longer applicable Reported issue: I-01	*If inflationBeneficiary is zero then pendingInflationBeneficiary is zero.*	*Original report*

katTokenCannotBeChangedOutsideInit rootCannotBeChangedOutsideInit	No longer applicable	*KatToken and root can only be changed by the init function.*	*Original report*
indexIsClaimedValueChange	No longer applicable	*indexIsClaimed(index) can only change false -> true*	*Original report*
rootSetterValueChange	No longer applicable	*rootSetter can only change from x -> address(0)*	*Original report*

Appendix 1: Late-stage revision at 494205c

As part of the mitigation review for the main scope, Certora reviewed fixes to the contracts in scope, specifically the PR #5, which at the time of this late-stage review had its head at commit 494205c.
At this stage, the key changes were the fixes to the findings reported, the removal of the `MerkleMinter` contract, and a rework of the permissioned roles in `KatToken`.
After reviewing these changes, Certora identified and reported the following additional findings:

L-03 Unbounded inflation allows admins to DoS minting

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: KatToken.sol	Status: Fixed in 9b348dd	

Description: The `KatToken` contract allows the `INFLATION_ADMIN` address to call its `changeInflation` function to adapt the inflation rate. While in the original scope, there was an upper limit to the inflation rate that could be configured.

Since this limit was removed, administrators can now set any arbitrarily high value.

Exploit Scenario: The address holding the `INFLATION_ADMIN` role can set an extremely high value for inflation, then trigger a distribution that would bring `distributedSupplyCap` close to `uint256.max`. Once the contract is at this stage, no more inflation can be distributed.

More generally, token holders could be reassured by the existence of a limit to how much their tokens can be diluted over time.

Recommendations: Consider reintroducing an upper limit - if flexibility drove the change, a limit that is higher than it used to be can be envisaged.

Customer's response: Fixed in 9b348dd.

Fix Review: Issue fixed.

I-06. Error-prone setLockExemption implementation

Description: The `KatToken.setLockExemption()` function allows the `LOCK_EXEMPTION_ADMIN` to toggle an override to the contract locking mechanism for a given address. Because this function does not take the desired status as input, but instead reverses the current stored setting, it is unnecessarily error-prone.

Recommendation: Consider adding a boolean parameter that allows the caller to define the desired exemption setting for the given address explicitly.

Customer's response: Fixed in e54329b.

Fix Review: Issue fixed.

I-07. Inconsistent behavior when renouncing roles with pending transfers

Description: There is an inconsistency in how different roles are renounced within the `KatToken` contract. Specifically:

- The `renounceInflationAdmin()` and `renounceInflationBeneficiary()` functions **revert** if there is a pending role transfer (i.e., `pendingRoleHolder[role] != address(0)`).
- In contrast, the `unlockAndRenounceUnlocker()` and `renounceLockExemptionAdmin()` functions **clear** the `pendingRoleHolder[role]` field and proceed with the renouncement.

Recommendation: Standardize the behavior across all role renouncement functions

Customer's response: Acknowledged. The different roles have different expected usage patterns, and the reported behaviour is an intentional decision.

I-08. `isUnlocked()` function fails to unlock at exact `unlockTime`

Description: The `isUnlocked()` function currently uses a strict greater-than check against `unlockTime`:

```javascript
function isUnlocked() public view returns (bool) {
    return (block.timestamp > unlockTime) || !locked;
}
```

This means that at the exact moment when `block.timestamp == unlockTime`, the contract is still considered locked, which contradicts the intuitive meaning implied by the function name `isUnlocked()`. Users would reasonably expect the contract to be unlocked once the specified `unlockTime` is reached.

Recommendation: Update the condition to be inclusive of `unlockTime`

Customer's response: Acknowledged. The current implementation provides a long-term gas optimization that is preferable.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.