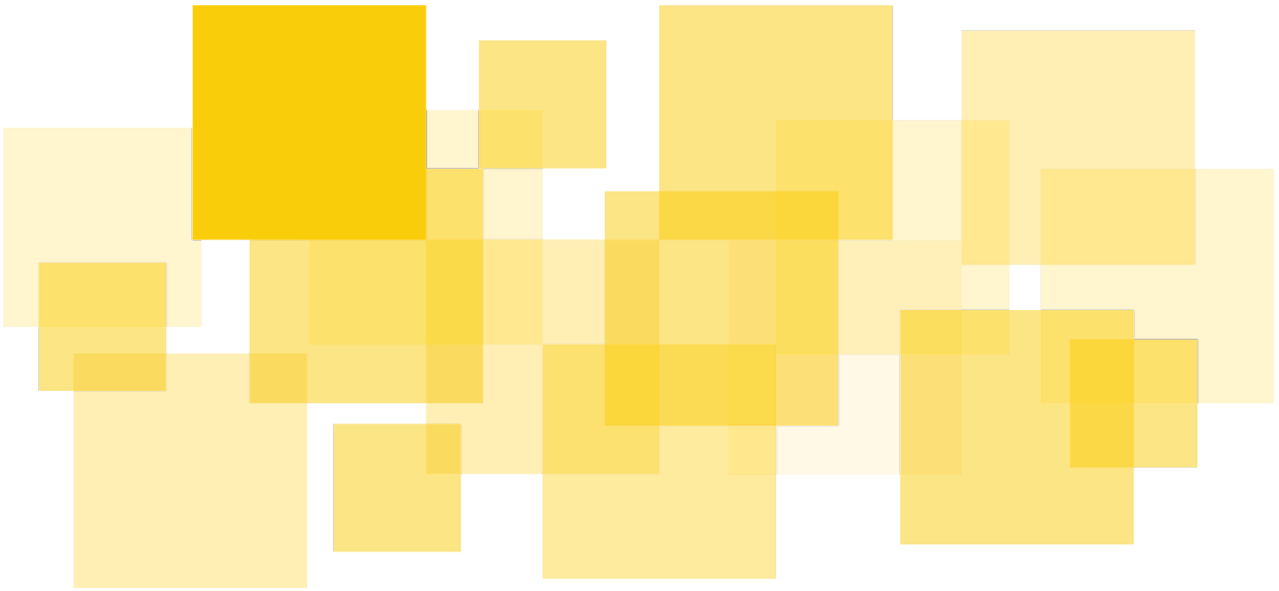
Security Audit Report

EspTokenV2 Ethereum

Delivered: November 13th, 2025



Prepared for Espresso Systems Inc. by

runtime verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks that otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis, and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carry substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims that impact a large quantity of funds.

Executive Summary

Espresso Systems engaged Runtime Verification Inc. to perform a security audit of the Espresso token design and smart contract implementation. The objective of the audit was to evaluate the business logic, Solidity code, and overall token mechanics to identify any potential vulnerabilities, misconfigurations, or behaviors that could lead to malfunction or exploitation.

The Espresso token serves as a core component within the broader Espresso Network — a decentralized sequencing infrastructure that provides a shared confirmation layer for Ethereum rollups through HotShot consensus. While the network as a whole enables multiple rollups to share Byzantine fault-tolerant sequencing and achieve verifiable finality via SNARK proofs on Ethereum L1, this audit focused exclusively on the token contracts that support the network's economic and governance mechanisms.

The reviewed components included the token's issuance and distribution logic, supply and transfer functions, and any privileged roles or administrative controls associated with the token's lifecycle. Particular attention was given to ensuring compliance with ERC standards, correct access control configuration, and resilience against known classes of smart contract vulnerabilities.

The token review was conducted over a calendar week, from November 5th to November 12th, as part of a broader engagement. Runtime Verification began with an initial walkthrough meeting to define the scope and understand the token's intended design, followed by a detailed manual review of the Solidity implementation. This review was complemented by formal property checking using Kontrol, our formal verification tool, to reason about key security and correctness properties across symbolic state transitions.

Scope

The scope of the audit is limited to the code contained in the GitHub repository provided by the client, which is public. Within this repository, specific files and contracts were highlighted as being in scope for this engagement. The repository, contracts, and commit information are detailed below:

EspressoSystems/espresso-network
https://github.com/EspressoSystems/espresso-network
Commit: 8b395dd

The following components were included in this audit:

- EspTokenV1
- EspTokenV2

The audit is limited to the artifacts listed above. Off-chain components, auto-generated code, deployment and upgrade scripts, and any client-side logic are excluded from the scope of this engagement.

Our security analysis is based on the following operational assumptions. If any of these assumptions are violated, the protocol's security guarantees may no longer hold, and additional review may be required.

- Trusted Admin Addresses
  All addresses holding admin roles are trusted, properly secured, and act in good faith.

Methodology

Although manual code review cannot guarantee the discovery of all possible security vulnerabilities, as noted in our Disclaimer, we followed a systematic approach to make this audit as thorough and impactful as possible within the allotted timeframe.

We rigorously analyzed the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and consistency between the logic and its implementation.

Findings presented in this report stem from a combination of:

- Manual inspection of the Solidity source code.
- Design-level reasoning about contract interactions.

This process was aided by our tools:

- ERCx - our platform, which checks the conformance of a contract to ERC(Ethereum Request for Comments) standards.
- Kontrol, which enables symbolic execution of Solidity code. Kontrol was used to run the existing test suite against symbolic inputs.

In addition to identifying bugs and vulnerabilities, we also reviewed edge-case handling and provided recommendations for code clarity and safety improvements.

Throughout the engagement, we held internal discussions among auditors to cross-review the findings and validate risk assessments.

EspTokenV2

EspTokenV2 serves as the staking and reward token for the Espresso Sequencer network. The contract extends EspToken(v1) through a UUPS proxy upgrade pattern, introducing controlled minting capabilities for validator reward distribution.

The primary change from the previous version is the `rewardClaim` contract reference, which is set during V2 initialization via `initializeV2()`. This address becomes the only minting authority, enforcing strict access control.

The EspToken(v1) is based on OpenZeppelin's implementation of ERC20Upgradeable (version 5.0.2), inheriting standard token functionality, as well as Initializable, OwnableUpgradeable, and UUPSUpgradeable features. The token contract updates the `_authorizeUpgrade()` function, which restricts contract upgrades to the `EspToken` owner. Additionally, it uses the SafeExitTimelock mechanism for ownership, underscoring its focus on simplicity due to its status as a straightforward ERC20 token, with minimal expected upgrade complexity.

The token contract does not implement a pausing mechanism. Instead, it is delegated to the RewardClaim contract, ensuring that users retain control of their assets in the event of an emergency.

Additionally, the developers used a frozen inheritance pattern during development, where each new version inherits the previous version of the contract, while the old version remains untouched.

A list of properties has been tested using the ERCx analyzer; the results have been attached in Appendix 1.

Additionally, we executed the test suites for EspToken and EspTokenV2 found under `contracts/tests` using Kontrol. The report is available on KaaS. No issues were discovered after running the symbolic execution tool.

Findings

This section contains all issues identified during the audit that could lead to unintended behavior, security vulnerabilities, or failure to enforce the protocol’s intended logic. Each issue is documented with a description, potential impact, and recommended remediation steps.

A01: Frozen inheritance pattern lacks an explicit documentation in `EspToken`

Severity: Low | Difficulty: High | Recommended Action: Document Prominently | Addressed by client

The EspToken and EspTokenV2 contracts are implemented using a frozen inheritance approach, where the new versions inherit the previous one while the parent contracts remain unchanged. However, this constraint is not enforced or documented throughout the code. This adds a risk that future maintainers may modify the parent contract storage layouts.

```
contract EspTokenV2 is EspToken {
    /// @notice Address of the RewardClaim contract authorized to mint tokens
    /// @notice Can only be set once, during initialization.
    address public rewardClaim;
```

If a future maintainer adds storage variables to `EspToken`, unaware of the frozen inheritance convention, those variables will occupy slot 0 and collide with EspTokenV2's `rewardClaim` storage in the proxy contract.

Recommendation

Document the frozen inheritance pattern in the source code of the `EspToken` contract, explicitly stating that parent contract storage layouts must never be modified.

Since all the dependencies of the token (`Initializable`, `ERC20Upgradeable`, `OwnableUpgradeable`, and `UUPSUpgradeable`) use the namespaced storage approach, consider applying the same pattern here for consistency and stronger collision prevention.

Status

The team documented this approach in both `EspToken` and `EspTokenV2` in the 4ed7cac commit.

Additionally, the client added a new test to the test suite (`NetworkStorageLayoutTest`) in 1e385b7. This ensures that the storage layout is maintained across versions by comparing deployed contract layouts from Etherscan against local compilation artifacts, catching any incompatibilities before the new contract gets deployed.

Informative Findings

This section includes observations that are not directly exploitable but highlight areas for improvement in code clarity, maintainability, or best practices. While not critical, addressing these can strengthen the system's overall robustness.

B01: Ownership can be renounced in upgradeable contracts

Severity: Informative | Difficulty: High | Recommended Action: Fix Code | Addressed by client

The `renounceOwnership()` function inherited from `OwnableUpgradeable` is not disabled in `EspToken`, `EspTokenV2`, and `RewardClaim`. This allows the owner to renounce ownership, leaving the contracts without an owner irrevocably.
As these are upgradeable contracts, this creates a risk that the `_authorizeUpgrade()` function can never be called, permanently locking the contract at its current implementation and preventing any future upgrades.

Recommendation

Override `renounceOwnership()` to revert, preventing accidental or malicious ownership renunciation.

```
error OwnershipCannotBeRenounced();

function renounceOwnership() public virtual override onlyOwner {
    revert OwnershipCannotBeRenounced();
}
```

Status

Addressed in 4e80f68. The `renounceOwnership` function has been overridden to revert in `EspTokenV2`, and new tests have been added to check its behavior.

B02: `EspToken._authorizeUpgrade` emits redundant event

Severity: Informative | Difficulty: High | Recommended Action: Fix Code | Addressed by client

In the `EspToken` contract, the `_authorizeUpgrade` function is overridden to emit a custom `Upgrade(address)` event.

```
    /// @notice only the owner can authorize an upgrade
    function _authorizeUpgrade(address newImplementation) internal virtual override
onlyOwner {
        emit Upgrade(newImplementation);
    }
```

However, the implementation of ERC1967Utils.upgradeToAndCall that is executed during `UUPSUpgradeable.upgradeToAndCall` already emits the standard `Upgraded(address)` event.

This has no impact on contract functionality; however, it incurs an additional gas cost and could confuse listeners who track the contract-emitted events.

Recommendation

Remove the duplicated `Upgrade(address)` event from the contract and from `_authorizeUpgrade` function.

Status

Addressed in 028fb63. The event has been removed. Additionally, the `EspTokenUpgradabilityTest.testUpgrade` test has been tweaked to check the emitted events.

Appendix 1 - ERCx report

This Appendix contains a table of all properties checked against the EspTokenV2 contract using fuzzing.
These have been extracted from the report generated by ERCx. Properties confirmed to pass are marked with a ✔️ symbol.

Id	Property	Status
1	All functions and events required by ERC20 standard are present and conformant.	✔
2	A tokenReceiver SHOULD NOT be able to call `transferFrom` of an amount more than her allowance from the tokenSender even if the tokenSender's balance is more than or equal to the said amount.	✔
3	A tokenReceiver SHOULD NOT be able to call `transferFrom` of an amount more than the tokenSender's balance even if the tokenReceiver's allowance from the tokenSender is less than the said amount.	✔
4	A tokenSender (which is also the `msg.sender`) SHOULD NOT be able to call `transfer` of an amount more than his balance.	✔
5	A tokenReceiver SHOULD NOT be able to call `transferFrom` of any positive amount from a tokenSender if the tokenSender did not approve the tokenReceiver previously.	✔
6	A `msg.sender` SHOULD NOT be able to call `transferFrom` of any positive amount from his/her own account to any tokenReceiver if the `msg.sender` did not approve him/herself prior to the call.	✔
7	A successful `approve` call of positive amount MUST emit the `Approval` event correctly.	✔
8	After a tokenApprover approves a tokenApprovee some positive amount via an `approve` call, any positive amount up to the said amount MUST be transferable by tokenApprovee via a `transferFrom` call, provided a sufficient balance of tokenApprover.	✔
9	After a tokenApprover approves a tokenApprovee some positive amount via an `approve` call, zero amount MUST be transferable by tokenApprovee via a `transferFrom` call, provided a sufficient balance of tokenApprover.	✔
10	Positive approved amount MUST be reflected in the allowance correctly.	✔
11	A successful `transfer` call of positive amount MUST emit the Transfer event correctly.	✔
12	A successful `transferFrom` call of positive amount MUST emit Transfer event correctly.	✔
13	A successful `approve` call of zero amount MUST emit the `Approval` event correctly.	✔
14	Zero approved amount MUST be reflected in the allowance correctly.	✔
15	A successful `transferFrom` of zero amount by any user other than the tokenSender, from and to different accounts, MUST emit a Transfer event correctly.	✔
16	A successful `transferFrom` call of zero amount by any user other than the tokenSender MUST be possible, from and to the same account.	✔
17	A successful `transferFrom` call of zero amount by any user other than the tokenSender MUST be possible, from and to different accounts.	✔
18	A successful `transferFrom` of zero amount by any user other than the tokenSender, from and to the same address, MUST emit a Transfer event correctly.	✔
19	A successful `transferFrom` call of zero amount by any user other than the tokenSender to the tokenSender MUST be possible.	✔
20	A successful `transferFrom` call of zero amount by the tokenSender herself MUST be possible.	✔
21	A successful `transferFrom` call of zero amount by the tokenSender herself to herself MUST emit a Transfer event correctly.	✔
22	A successful `transferFrom` call of zero amount by the tokenSender herself to someone MUST emit a Transfer event correctly.	✔
23	A successful `transfer` call of zero amount to another account MUST emit the Transfer event correctly.	✔
24	A successful `transfer` call of zero amount to another account MUST be possible.	✔
25	A successful `transfer` call of zero amount to self MUST emit the Transfer event correctly.	✔
26	A successful `transfer` call of zero amount to self MUST be possible.	✔
27	The zero address SHOULD NOT have any token from the contract.	✔
28	Contract owner cannot control balance by using minting overflow (via `mint(address,uint256)` or `mintToken(address,uint256)` or `issue(address,uint256)`).	✔

Id	Property	Status
29	A `msg.sender` SHOULD be able to retrieve his/her own balance.	✔
30	A `msg.sender` SHOULD be able to retrieve balance of an address different from his/hers.	✔
31	A successful call of `approve` of any amount to the zero address SHOULD NOT be allowed.	✔
32	If consecutive calls of `approve` function of positive-to-positive amounts can be called, then the allowance is set to the right amount after the second call.	✔
33	Consecutive calls of `approve` function of positive-to-zero amounts CAN be called and the allowance is set to the right amount after the second call.	✔
34	Consecutive calls of `approve` function of zero-to-positive amounts CAN be called and the allowance is set to the right amount after the second call.	✔
35	Consecutive calls of `approve` function of zero-to-zero amounts CAN be called and the allowance is set to the right amount after the second call.	✔
36	It is possible to deal the intended amount of tokens to dummy users for interacting with the contract.	✔
37	The `transferFrom` function DOES NOT have the potential to take fees.	✔
38	The `transferFrom` function DOES NOT take fees at test execution time.	✔
39	The `transfer` function DOES NOT have the potential to take fees.	✔
40	The `transfer` function DOES NOT take fees at test execution time.	✔
41	Minting to the zero address should not be possible (via `mint(address,uint256)` or `mintToken(address,uint256)` or `issue(address,uint256)`). Minting to this address should fail.	✔
42	User balance should be updated correctly after minting (via `mint(address,uint256)` or `mintToken(address,uint256)` or `issue(address,uint256)`).	✔
43	Total supply should be updated correctly after minting (via `mint(address,uint256)` or `mintToken(address,uint256)` or `issue(address,uint256)`).	✔
44	Minting zero token (via `mint(address,uint256)` or `mintToken(address,uint256)` or `issue(address,uint256)`) should not change the balance of the account target of the mint.	✔
45	Minting zero token (via `mint(address,uint256)` or `mintToken(address,uint256)` or `issue(address,uint256)`) should not change the total supply.	✔
46	Multiple calls of `transferFrom` SHOULD NOT be allowed once allowance reach zero even if the tokenSender's balance is more than the allowance.	✔
47	Multiple `transfer` calls of positive amounts are ALLOWED given that the sum of the transferred amounts is less than or equal to the tokenSender's balance.	✔
48	Multiple `transferFrom` calls of positive amounts are ALLOWED given that the sum of the transferred amounts is less than or equal to the tokenSender's balance and approvals are given by the tokenSender.	✔
49	Self approval of positive amount is ALLOWED and the allowance is correctly updated.	✔
50	Self approval and call of `transferFrom` from its own account of positive amount is ALLOWED.	✔
51	Self `transfer` call of positive amount is ALLOWED and SHOULD NOT modify the balance.	✔
52	A tokenReceiver CAN call `transferFrom` of the tokenSender's total balance amount given that tokenSender has approved that.	✔
53	A `msg.sender` CAN call `transfer` of her total balance amount to a tokenReceiver and the balances are modified accordingly.	✔
54	A `transferFrom` call of any positive amount to the zero address SHOULD revert.	✔
55	A `transfer` call of any positive amount to the zero address SHOULD revert.	✔
56	The contract's `totalSupply` variable SHOULD NOT be altered after `transfer` is called.	✔
57	The contract's `totalSupply` variable SHOULD NOT be altered after `transferFrom` is called.	✔

Id	Property	Status
58	The value of variable totalSupply, if it represents the sum of all tokens in the contract should not overflow when the sum of tokens changes via minting (via `mint(address,uint256)` or `mintToken(address,uint256)` or `issue(address,uint256)`).	✔
59	A successful call of `transfer` DOES NOT update the balance of users who are neither the tokenSender nor the tokenReceiver.	✔
60	A successful `transferFrom` of any positive amount MUST decrease the allowance of the tokenSender by the transferred amount.	✔
61	A successful call of `transferFrom` DOES NOT update the balance of users who are neither the tokenSender nor the tokenReceiver.	✔
62	A `approve` call of any positive amount SHOULD revert if the tokenSender is the zero address.	✔
63	A `transfer` call of any positive amount SHOULD revert if the tokenSender is the zero address.	✔
64	Multiple calls of `transfer` of zero amount are ALLOWED.	✔
65	Multiple calls of `transferFrom` of zero amount are ALLOWED.	✔
66	Self approval of zero amount is ALLOWED and the allowance is correctly updated.	✔
67	Self approval and call of `transferFrom` from its own account of zero amount is ALLOWED.	✔
68	Self `transfer` call of zero amount is ALLOWED and SHOULD NOT modify the balance.	✔
69	A tokenReceiver CAN call `transferFrom` of the tokenSender's total balance amount of zero.	✔
70	A `msg.sender` CAN call `transfer` of her total balance of zero to a tokenReceiver and the balances are not modified.	✔
71	A `transferFrom` call of zero amount to the zero address SHOULD revert.	✔
72	A `transfer` call of zero amount to the zero address SHOULD revert.	✔