Security Audit Report

Espresso Systems Ethereum

Delivered: November 24th, 2025

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 conduct a security audit of the Espresso Network staking component design and the associated smart contract code. The objective was to review the business logic and implementation in Solidity and identify any issues that could cause the system to malfunction or be vulnerable to exploitation.

Espresso Systems is a decentralized sequencing infrastructure that provides a shared confirmation layer for Ethereum rollups through HotShot consensus. The network enables multiple rollups to leverage the same Byzantine fault-tolerant sequencing infrastructure, providing sub-second transaction confirmations with verifiable finality through SNARK proofs submitted to Ethereum L1. It operates as a marketplace for rollup sequencing, where each rollup assigns itself a namespace identifier to extract its transactions from the shared Espresso ledger, along with succinct proofs of inclusion. Espresso integrates with both ZK and optimistic rollups through light client contracts that verify state updates on L1, with an escape hatch mechanism allowing rollups to fall back to backup sequencers if the Espresso consensus loses liveness. The network is designed to provide fast, verifiable, and decentralized transaction ordering for the rollup-centric Ethereum ecosystem while maintaining composability across participating chains.

The audit was conducted over three calendar weeks, from November 5th to November 19th. Runtime Verification conducted an initial walkthrough call with the client to understand the components in scope and their design, followed by a focused manual review of the Solidity implementation. We used Kontrol, our formal verification tool, to support this process by specifying and checking 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:

- EspTokenV2
- StakeTableV2
- RewardClaim
- RewardMerkleTreeVerifier
- OpsTimelock
- SafeExitTimelock

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 contracts, validating security-critical properties to ensure the absence of loopholes and to check consistency between the intended 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.

Network Components Summary

In this section, we review the components of the Espresso Network that we examined during our engagement. We then detail the patterns and architecture employed across the codebase.

Key Contracts Overview

During the engagement, we have reviewed the following components that implement the key functionality of the protocol.

Additionally, these components utilize a frozen inheritance pattern to manage storage layout and ensure upgrade safety. In this pattern, once a contract version is deployed (for example, StakeTable V1), its source code remains unchanged and serves as an immutable foundation for future versions. Subsequent upgrades (such as StakeTableV2 and EspTokenV2) inherit from this frozen parent and only introduce new functionality or storage variables at the end of the storage layout.

EspToken

EspTokenV2 is the token used for staking and reward distribution within 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 overrides the `_authorizeUpgrade()` function, which restricts contract upgrades to the `EspToken` owner. Additionally, it uses the SafeExitTimelock mechanism for ownership, underscoring the contract's 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.

A list of properties was 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.

StakeTable

StakeTableV2 manages the validator registration and token delegation within the Espresso network. The architecture is event-driven, where Layer 1 emits state changes that are then processed by the off-chain sequencer node to execute the required operations. Notably, this version lacks a slashing mechanism, as that feature is planned to be added in a future update.

One of the main improvements in StakeTableV2 compared to the initial version is the introduction of dual signature validation for critical operations. Specifically, the functions `registerValidatorV2()` and `updateConsensusKeysV2()` require both BLS and Schnorr signatures on-chain. The contract enforces Schnorr key uniqueness through a mapping called `schnorrKeys`, which prevents the reuse of keys among different validators and generates a `SchnorrKeyAlreadyUsed` error if such reuse is attempted. Functions from version one that are now deprecated will revert with a `DeprecatedFunction()` error.

Validators can adjust their commission rates via the `updateCommission()` function, which includes built-in rate limiting to ensure that increases do not exceed 5% within a rolling 7-day window, by default. Commission values are tracked on Layer 1, but the actual enforcement occurs off-chain during the reward distribution process to enhance gas efficiency. The contract owner is granted the authority to modify the parameters of these rate limits using `setMinCommissionUpdateInterval()` (with permissible intervals ranging from 1 second to 365 days) and `setMaxCommissionIncrease()` (allowing increases from 1 to 10,000 basis points).

StakeTableV2 includes an emergency pausability feature. The `PAUSER_ROLE` role is empowered to pause user-facing functions, which include actions such as delegating, undelegating, claiming withdrawals, and updating commissions. However, certain governance functions remain operational even during a pause: `updateExitEscrowPeriod` , `setMinCommissionUpdateInterval()` , and `setMaxCommissionIncrease()` are all non-pausable, ensuring continuity in critical governance activities.

Another improvement from the StakeTable(v1) is the introduction of active stake tracking using an `activeStake` variable, which tracks the total amount of stake that has not yet been exited. This variable is updated during delegation and undelegation actions to keep accurate records. Also, the handling of deferred validator exits has been improved: the `delegatedAmount` now decreases during the `claimValidatorExit()` process rather than during `deregisterValidator()` .

The contract also permits adjustments to the escrow period, with the owner able to modify the `exitEscrowPeriod` within defined limits that range from a minimum of 15 blocks to a maximum of 14 days.

Migrations from StakeTable (v1) to StakeTableV2 are facilitated through the `initializeV2()` function, which acts as a reinitializer. It allows the input of an `initialCommissions` array to facilitate the migration of validator commissions from the first version. Importantly, the function `_initializeCommissions()` includes validation checks to ensure that commissions have not been previously initialized. Furthermore, the `initialActiveStake` parameter must not exceed the contract's token balance; if it does, an `InitialActiveStakeExceedsBalance` error will occur. To assist with this migration, off-chain tooling is available, which retrieves state data from version one by examining relevant events.

RewardClaim

The RewardClaim contract is created to help users claim their ESP token rewards by providing cryptographic Merkle proofs of the rewards they've earned over time. It verifies these reward claims by reconstructing the reward Merkle root and comparing it with the stored auth root in the Light Client, before minting the corresponding ESP tokens.

To track and prevent double claiming of rewards, the contract maintains a mapping that tracks the total lifetime rewards claimed by each user. Additionally, a daily limit system is implemented. It checks the current day based on the block timestamp and resets the claims if a new day is detected. Additionally, it increments the total amount claimed for the day and will revert any transactions if the limit has been exceeded.

The `IRewardClaim` interface is designed to communicate to partners what the interface would look like for them to perform reward claims. So it only defines the `claimRewards` functionality and associated events and errors.

RewardMerkleTreeVerifier

The RewardMerkleTreeVerifier library implements a binary Merkle tree verifier with a depth of 160 levels, matching Ethereum address length (20 bytes). It uses EVM-native `keccak256` hashing with double-hashing of leaf values as a domain separator instead of explicit domain separation tags.

The `computeRoot` function reconstructs the Merkle root by using a claimant's address as the key, their total rewards as the value, and a proof array with 160 sibling hashes for each tree level. Starting from the leaf (which is the double-hashed reward amount), it moves upward through the tree by extracting each bit of the address with right-shift operations to decide the branch direction, then hashing the current node with its sibling using `_hashInternal`.

The computed root serves as the reward commitment that RewardClaim.sol hashes with seven authentication inputs before comparing against the Light Client's authRoot, enabling cryptographic verification that a specific address has earned specific lifetime rewards without revealing the entire reward state tree on-chain.

OpsTimelock and SafeExitTimelock

The OpsTimelock and SafeExitTimelock contracts provide time-delayed governance through OpenZeppelin's TimelockController implementation, enforcing role-based access control.s. Both contracts follow the standard schedule-execute pattern, where proposals are assigned unique operation IDs and tracked through states (Unset, Waiting, Ready, Done) to prevent early or duplicate execution.

The main architectural difference between the two is in their delay periods and scope of ownership. OpsTimelock uses shorter delays (~1 day) for critical infrastructure that needs quick response capabilities (FeeContract, LightClient, StakeTable), while SafeExitTimelock ensures much longer delays (~14 days) for user-facing contracts (EspToken, RewardClaim), where users require extended exit windows to withdraw tokens or claim rewards before upgrades are enforced.

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 creates a risk that future maintainers may modify the parent contract's storage layout.

```
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.

A02: Missing initialization of `PausableUpgradable` for StakeTableV2

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

The `initializeV2()` function of the `StakeTableV2` contract does not call the `__Pausable_init()` function. Therefore, the `PausableUpgradeable` functionality is not properly initialized.

```
function initializeV2(
    address pauser,
    address admin,
    uint256 initialActiveStake,
    InitialCommission[] calldata initialCommissions
) public onlyOwner reinitializer(2) {
    __AccessControl_init();

    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(PAUSER_ROLE, pauser);
```

Recommendation

Add the missing initializer by invoking `__Pausable_init()` inside `initializeV2()`.

Status

Addressed in fd4c067.

A03: Missing checks for schnorrKeys in `updateConsensusKeysV2()`

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

In `StakeTableV2.updateConsensusKeysV2()`, the function does not verify the length of `schnorrKey`, nor does it update its value in the `schnorrKeys` mapping.

```
        bytes memory message = abi.encode(validator);
        BLSSig.verifyBlsSig(message, blsSig, blsVK);

        blsKeys[_hashBlsKey(blsVK)] = true;

        emit ConsensusKeysUpdatedV2(validator, blsVK, schnorrVK, blsSig, schnorrSig);
    }
```

These actions are present in the `registerValidatorV2` function.

```
    function registerValidatorV2(
    ...
        // ensure that the schnorrSig is the correct length
        if (schnorrSig.length != 64) {
            revert InvalidSchnorrSig();
        }
        ...
        schnorrKeys[_hashSchnorrKey(schnorrVK)] = true;
```

Recommendation

Add the missing actions to the `updateConsensusKeysV2` function.

Status

Fixed in commit e2d3c52.

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`, `StakeTable`, and `RewardClaim`. This allows the owner to renounce ownership, leaving the contracts without an owner permanently.
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

For `EspToken`, this was addressed in 4e80f68. The `renounceOwnership` function has been overridden to revert in `EspTokenV2`, and new tests have been added to check its behavior.

For `RewardClaim`, `StakeTable`, and remaining contracts, this was addressed in 037c62c.

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 the `_authorizeUpgrade` function.

Status

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

B03: `StakeTable.delegate()` function does not follow the checks-effects-interactions pattern

Severity: Informative Difficulty: High Recommended Action: Fix Code

Addressed by client

The `StakeTableV2.delegate()` function calls `super.delegate()` , which performs certain state updates **after** a token transfer.

```
    function delegate(address validator, uint256 amount) public virtual override
whenNotPaused {
        super.delegate(validator, amount);
        activeStake += amount;
    }
```

Although this execution path ultimately uses a token under the protocol's control, the contract still relies on ERC20-compatible transfer semantics. It might be advisable to rewrite the function in V2 instead of calling `super` .

Status

Function was rewritten to follow the checks-effects-interactions pattern in 4ac21c6.

B04: Missing bound control for escrow period update

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

In `StakeTableV2.updateExitEscrowPeriod()` , boundary values for the escrow period are set.

```
        uint64 minExitEscrowPeriod = lightClient.blocksPerEpoch() * 15; // assuming 15
seconds per
            // block
        uint64 maxExitEscrowPeriod = 86400 * 14; // 14 days
```

The lower bound is dynamically set over a function call to `lightClient.blocksPerEpoch()` . If `lightClient.blocksPerEpoch()` returns a value where `blocksPerEpoch() * 15 > 86400 * 14; // 14 days` , the minimum would exceed the maximum, causing all calls to `updateExitEscrowPeriod()` to revert with `ExitEscrowPeriodInvalid()` .

Recommendation

Add validation to ensure `minExitEscrowPeriod < maxExitEscrowPeriod` , or make the maximum dynamic based on light client parameters.

Status

Addressed in 8228b3a.

B05: Reward distribution does not consider fairness

Severity: Informative | Difficulty: High | Recommended Action: Document Prominently

Not addressed by client

The `RewardClaim` contract enforces daily reward claim limits, operating on a first-come, first-served basis. After the daily limit is reached, unrewarded participants must wait until the next day, but they risk missing the subsequent period if the limit is reached again.

The client acknowledged this and stated that the daily limit is unlikely to be exceeded and that accepting this risk is acceptable. However, it may be a significant risk, as token valuation can fluctuate significantly within 24 hours.

Status

The client has documented this in 2170cd8.

B06: Possible gas saving by using EVM local scratch memory

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

In `RewardMerkleTreeVerifier._hashInternal()`, the use of the free memory reservation as in `let ptr := mload(0x40)` is unnecessary.

```
assembly {
    let ptr := mload(0x40) // Get free memory pointer
    mstore(ptr, left) // Store left (32 bytes)
    mstore(add(ptr, 0x20), right) // Store right (32 bytes)
    hash := keccak256(ptr, 0x40) // Hash 64 bytes
}
```

Recommendation

It should be possible to utilize the 64-byte scratch space at `0x00` and save additional gas, as detailed here.

```
mstore(0x00, left)
mstore(0x20, right)
hash := keccak256(ptr, 0x40)
```

It should be noted that `keccak256` has its own opcode, hence it would not use local memory.

Status

Addressed in 462fbcd.

B07: Unused error in RewardMerkleTreeVerifier

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

The RewardMerkleTreeVerifier library declares a custom error `InvalidProofLength()` that is never used anywhere in the codebase. The proof length in the `computeRoot` function is enforced at compile-time through Solidity's type system by using a fixed-size array parameter (`bytes32[160]`). This makes runtime validation unnecessary.

```
library RewardMerkleTreeVerifier {
    error InvalidProofLength();

    function computeRoot(address key, uint256 value, bytes32[TREE_DEPTH] memory proof)
        internal
        pure
        returns (bytes32)
    {
```

Status

The error was removed in 462fbcd.

B08: Commission and Basis Points hardcoded as 10000

Severity: Informative · Difficulty: Low · Recommended Action: Fix Code

Addressed by client

The value `10000` , representing 100% in basis points, is hardcoded throughout the codebase without a named constant. This reduces code readability and maintainability, as the semantic meaning (basis points denominator) is not immediately clear from the numeric literal.

Recommendation

Define a contract-level constant in both `StakeTableV2` and `RewardClaim` contracts.

Status

Addressed in 04107be.

B09: `RewardClaim._authorizeUpgrade` emits redundant event

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

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

```
    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 037c62c.

B10: No Zero Checks in RewardMerkleTreeVerifier.computeRoot()

Severity: Informative Difficulty: High Recommended Action: Document Prominently
Addressed by client

The `RewardMerkleTreeVerifier.computeRoot()` function lacks explicit zero-value checks for `key` and `value` parameters. However, this is intentional library design as validation occurs at the application layer: `RewardClaim.claimRewards()` validates `lifetimeRewards != 0` before invoking `_verifyAuthRoot()`, and `msg.sender` (used as key) cannot be `address(0)` in standard execution.

```
    function claimRewards(uint256 lifetimeRewards, bytes calldata authData)
{
...
        require(lifetimeRewards != 0, InvalidRewardAmount());
...
        require(_verifyAuthRoot(lifetimeRewards, authData), InvalidAuthRoot());
}
```

```
    function _verifyAuthRoot(uint256 lifetimeRewards, bytes calldata authData)
{
...
            RewardMerkleTreeVerifier.computeRoot(msg.sender, lifetimeRewards, proof);
...
}
```

Recommendation

No remediation required.

Status

This was noted in 462fbcd.

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.	✓