



Security Audit Report

FxHash Token Security Audit Ethereum

Delivered: February 3, 2025

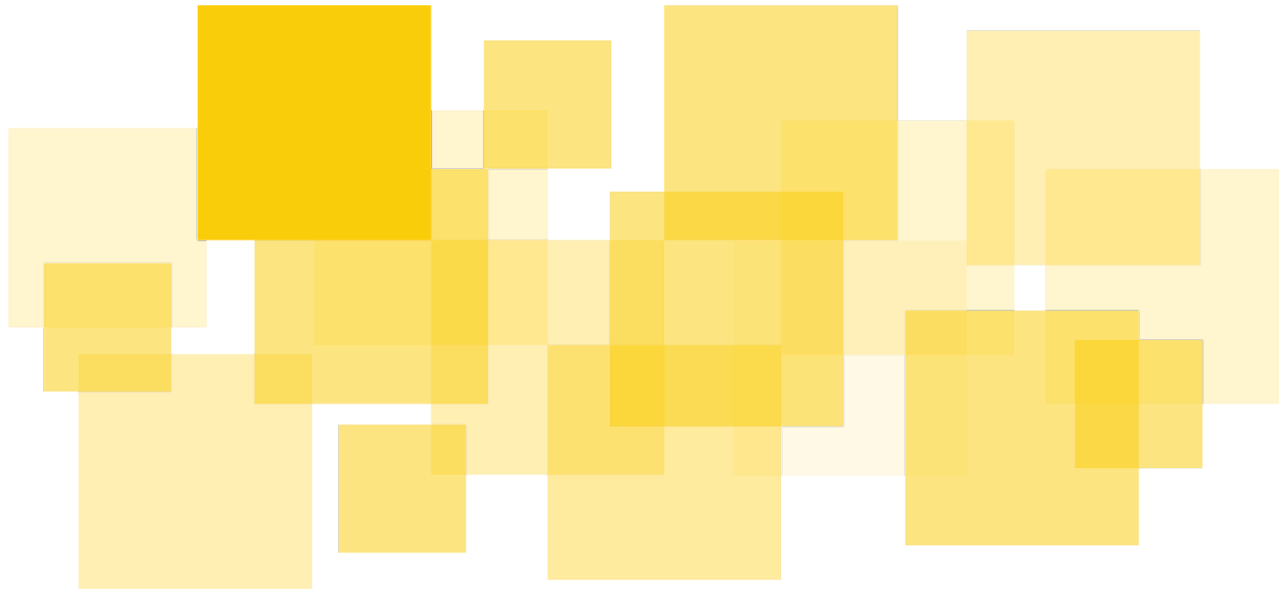




Table of Contents

- [Disclaimer](#)
- [Summary and Scope](#)
- [Methodology](#)
- [System Description](#)
- [Key Properties](#)
 - [Restrictions on Operations](#)
 - [Accounting and Balances](#)
- [Mathematical Model and Proofs](#)
 - [Vesting Schedule](#)
 - [Merkle Proof](#)
 - [No Double Claim](#)
- [Informative Findings](#)
 - [\[F01\] Temporary Synchronization Delays in FxToken and FxTokenL2 Bridge](#)
 - [\[F02\] Insufficient Event Emissions for Admin Actions](#)
 - [\[F03\] Centralization Risks in TezAirdrop](#)



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. 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. 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. 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.

Blockchain technology is still a nascent software arena, and any related implementation and public offering carries substantial risk.

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



Summary and Scope

Not addressed by client

The security audit of the FxHash Token was conducted by [Runtime Verification, Inc.](#) from January 10, 2025, until January 23, 2025. The objective of the review was to assess the overall architecture, pinpoint potential risks, and offer recommendations for improvements in functionality, security, and efficiency. The audit explores potential vulnerabilities and suggests security improvements based on the code's structure, logic, and interaction patterns.

The targets of the audit are the smart contract source files at git-commit-id [f8ca216e6274e7e33fbbba5cc36cf166ba34e0a8](#).

The audit focused on the following core contracts and interfaces:

- `FXToken`
- `FxVestingEscrow`
- `FxVestingFactory`
- `FxAirdrop`
- `FxTokenL2`
- `TezAirdrop`
- `OptimismMintableERC20`
- `IFxToken`
- `IFxVestingFactory`
- `IFxAirdrop`
- `ITezAirdrop`
- `IFxAirdropEventsAndErrors`
- `IFxVestingFactoryEventsAndErrors`
- `ITezAirdropEventsAndErrors`

This report presents a detailed review of the system's smart contracts. These contracts enable token distribution and management, including key functionalities such as vesting and airdrops.

The audit involved a meticulous examination of individual contracts and their interactions within the broader ecosystem. This analysis focused on assessing the functionality, identifying security properties, and pinpointing potential vulnerabilities.



Methodology

Although the manual code review cannot guarantee the finding of all possible security vulnerabilities, as mentioned in the [Disclaimer](#), we have followed a rigorous approach to maximize its effectiveness. We began by defining the intended behavior and invariants of the contracts under review. We then meticulously compared the actual implementation against these expectations, identifying discrepancies between the intended behavior and the actual implementation. We also carefully checked if the code is vulnerable to [known security issues and attack vectors](#). Finally, we summarized the key security properties that were successfully proven during the review process.



System Description

The system's smart contracts are meticulously designed to handle token-related operations, including minting, burning, vesting, airdrops, and bridging across Layer 1 (L1) and Layer 2 (L2) networks. Together, they form a robust framework for secure token distribution, seamless claim validation, and efficient cross-chain functionality.

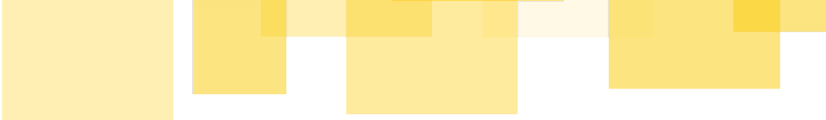
The system comprises the following key contracts, each fulfilling a distinct role:

- **FxToken**: An ERC20 token contract used for asset management.
- **FxVestingEscrow**: Manages token vesting and distribution over a specified period, including a cliff and linear vesting logic.
- **FxVestingFactory**: A factory contract that deploys instances of VestingEscrow to individual users.
- **FxTokenL2**: The layer2 version of the FxToken used in an optimized Layer 2 environment (e.g., Optimism).
- **FxAirdrop**: Manages the airdrop of tokens to eligible addresses, leveraging Merkle Trees for efficient, off-chain claim validation.
- **TezAirdrop**: This is a specialized version of the airdrop contract that validates claims using EIP712 signatures and Tezos wallet addresses.

FxToken and FxTokenL2 Contracts

The FxToken contract is an ERC20-compliant token deployed on L1, serving as the primary representation of the system's token. It supports administrative controls, including minting, burning, and pausing functionalities. FxToken incorporates role-based access control with `ADMIN_ROLE`, `MINTER_ROLE`, and `PAUSER_ROLE` to ensure only authorized actors can perform critical operations. For example, minting and burning tokens are restricted to specific roles to prevent misuse.

The FxTokenL2 contract is the L2 counterpart of FxToken. It implements token minting and burning functionality, which are directly triggered by the bridge to maintain a 1:1 peg with the L1 token. The contract uses a deterministic bridge mechanism to synchronize minting and burning actions between L1 and L2. Pausing functionalities, bypass addresses, and administrative role-based controls are also integrated into this contract.



The bridge leverages a deterministic mechanism to synchronize minting and burning actions. While transaction finalization delays between L1 and L2 are inherent due to network conditions, these delays do not compromise the integrity of the token peg. The sequencer ensures that synchronization remains reliable and eventual consistency is maintained. The immutable nature of the FxTokenL2 contract ensures that no unauthorized changes can affect the bridge's operations.

Airdrop Contracts: FxAirdrop and TezAirdrop

These contracts handle airdrop operations for distributing tokens to eligible participants using a Merkle Tree for claim validation.

The FxAirdrop supports token airdrops for Ethereum wallets. Each user's claim is represented as a leaf in the tree containing their address and allocation. Claims are validated against a Merkle Tree, ensuring users can only claim the amount allocated. The key features include:

- **Merkle Proof Validation:** Verify users' proof against the Merkle root to ensure that only eligible users can claim tokens.
- **Claim Tracking:** A `hasClaimed` mapping is used to prevent double claims.
- **Admin Functions:** Allows updating the Merkle root and lock timestamps for flexible management.

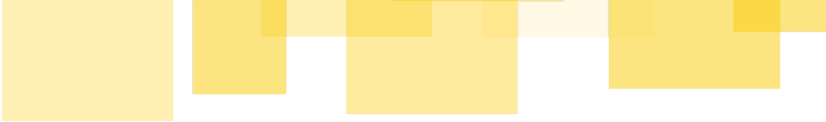
The TezAirdrop contract extends the functionality of FxAirdrop to include claims for Tezos wallets. In addition to Merkle proof validation, it uses EIP-712 signatures to verify ownership of the Tezos wallet. This additional layer ensures that claims are authenticated.

Both airdrop contracts employ mechanisms to track claimed tokens (`hasClaimed` mapping) and restrict further claims after the initial redemption. Admin functionalities include updating the Merkle root and claim lock timestamps.

Vesting Contracts: FxVestingFactory and FxVestingEscrow

The vesting system is built to gradually release tokens to beneficiaries over a defined period, ensuring controlled distribution and preventing premature access.

The FxVestingFactory serves as a deployment factory for FxVestingEscrow wallets, enabling the creation of custom vesting schedules for token allocations. Each vesting schedule is defined by parameters such as the start time, duration, and cliff period. The cliff period establishes a



time window during which no tokens are released, while the duration determines the overall vesting timeline. After the cliff, tokens are released linearly based on elapsed time until the full allocation is vested.

FxVestingEscrow enforces these parameters at the individual wallet level, ensuring beneficiaries receive tokens as per the defined schedule. The contracts ensure that all state updates occur before making external calls, minimizing risks such as reentrancy. Additionally, they use immutable logic to prevent unauthorized alterations to vesting parameters once set.

Security and Governance

The system employs several security mechanisms:

- **Role-Based Access Control:** This ensures that administrative functions (e.g., minting, pausing, and Merkle root updates) are restricted to specific roles.
- **Multi-Signature Governance:** Critical roles, such as those controlling token minting or pausing, are secured via multi-signature wallets in practice, mitigating centralization risks.
- **Replay Protection:** Both airdrop contracts use mappings (e.g., `hasClaimed`) to prevent double claims or signature reuse.
- **Immutable Contracts:** FxToken, FxTokenL2, and FxVestingEscrow are non-upgradeable, ensuring stability and resistance to unauthorized modifications.



Key Properties

This section presents properties that must be guaranteed by the implementation of the above contracts that comprise the system.

Restrictions on Operations

FxToken

- Transfers, minting, and burning must not occur while the contract is paused.
- Pausing and unpausing can only be executed by an address with the `PAUSER_ROLE`.
- Only addresses with the `MINTER_ROLE` can mint tokens.
- Token burning can only be performed by the token owner or approved spenders.
- Token transfer can only be performed by the token owners or approved spenders.
- The role hierarchy are strictly enforced: `DEFAULT_ADMIN_ROLE` > `ADMIN_ROLE` > `MINTER_ROLE/PAUSER_ROLE`.
- The total supply must accurately reflect all minting and burning events.

FxVestingEscrow

- Tokens cannot be released before the cliff period.
- Tokens cannot be unlocked or withdrawn before the vesting period ends.
- A user cannot withdraw more tokens than the vested amount at any given point in time.
- Only the designated beneficiary can claim released tokens.
- Released amounts must be correctly tracked and follow the defined linear vesting schedule.
- Vested amounts must be accurately calculated for any timestamp.
- Ownership transfer of the vesting contract should not impact the established vesting schedule.

FxVestingFactory

- Tokens must be transferred correctly to newly created vesting contracts.
- Vesting contract creation can only be initiated by the owner.
- Must track the deployments of all vesting contracts.



FxTokenL2

- Only the designated bridge contract can mint or burn tokens.
 - The contract must maintain a 1:1 backing ratio with the corresponding L1 token.
 - Transfers, minting, and burning must not occur while the contract is paused.
 - Permissions to bypass the paused state must be correctly enforced, preventing unauthorized unpausing.
- The contract must support automatic approval for allowances using the Permit2 standard.

FxAirdrop

- Each claim address can only be claimed once.
- Claims must be valid with a correct Merkle proof and cannot exceed the allocated amount.
- Withdrawals can only be initiated by the owner.
- Withdrawals must be performed after the defined lock period.
- All claims must be signed by the designated `signer`.
- Only the admin can update the Merkle root or perform withdrawals.

TezAirdrop

- Claims can only be made if the signature is valid and the Merkle proof is correct.
- Each Tezos wallet can only be claimed once.
- The claimed amount must not exceed the allocated amounts.
- Claims and withdrawals must adhere to the defined lock period.
- Only the admin can withdraw funds.
- Only the admin can update the Merkle root.

Accounting and Balances

- The total balance of tokens within the token contracts must equal the sum of all users' vested amounts.
- A user cannot withdraw more tokens than their vested amount.
- Total claims must not exceed the total token allocation specified in the Merkle tree.
- Any remaining unclaimed tokens must remain within the contract until explicitly withdrawn by the designated admin.



Additional Considerations

- Signatures must be unique and non-replayable for each claim.
- Merkle proof verification must correctly validate claims and reject invalid or bypassed proofs.



Mathematical Model and Proofs

This section provides mathematical models and formal proofs to ensure the correctness and integrity of the system's core properties and invariants. The models describe the expected behavior of critical mechanisms, while the proofs demonstrate that the implementation adheres to these expectations.

Vesting Schedule

The Vesting Schedule determines how tokens allocated to a beneficiary are gradually unlocked over a predefined period. It ensures that tokens are not immediately available to the beneficiary, providing a mechanism for gradual token distribution, often used in team allocations, investor rewards, or incentive structures.

Key components of the vesting schedule:

- **Cliff period:** The initial period where no tokens are released.
- **Vesting period:** The total time over which tokens are released linearly after the cliff period.

The following invariant must hold for the vesting schedule:

Invariant The number of tokens released at any time (t) satisfies: $\forall t. 0 \leq R(t) \leq M$.

This invariant ensures no tokens are released prematurely and no over-release occurs.

Mathematical Model

Let:

- T be the total vesting period, i.e., the time span after the cliff period during which tokens will be unlocked.
- C be the cliff period, i.e., the amount of time after which tokens begin to vest.
- M be the total number of tokens allocated to the beneficiary.
- t be the current timestamp.
- $R(t)$ be the number of tokens released by time t .

The formula for the release of tokens $R(t)$ is as follows:

$$R(t) = \begin{cases} 0, & \text{if } t < C \\ \frac{M}{T-C} \times (t - C), & \text{if } C \leq t \leq T \\ M, & \text{if } t > T \end{cases}$$

where:

- For $t < C$, no tokens are released (i.e., during the cliff period).

- For $C \leq t \leq T$, the tokens are released linearly. The number of tokens released grows with time, at a rate proportional to $\frac{M}{T-C}$.
- All tokens have been released for $t > T$, and the vesting is complete.

Proof

This section proves that the vesting schedule formula correctly releases the allocated tokens M over the entire vesting period $[C, T]$ and prevents premature releases.

1. Token Release Begins after Cliff:

The formula states that for $t < C$, $R(t) = 0$, meaning no tokens are released. This confirms that the cliff period is enforced correctly. Thus, the contract ensures that tokens are not released before the cliff.

For $t < C$, $R(t) = 0$ (No tokens released before cliff)

2. Linear Release of Tokens after Cliff:

The formula for $C \leq t \leq T$ ensures a linear release of tokens. The rate of release is given as the formula: $\text{Release rate} = \frac{M}{T-C}$. Since the formula is linear, for every time unit after C , the released amount increases by $\frac{M}{T-C}$.

- When $t = C$, the amount released is 0 tokens, and at time T , the total amount released is exactly M .
- Therefore, at time $t = T$, the formula will release $R(T) = M$, which is the full amount allocated.

3. All Tokens Released After Vesting Period:

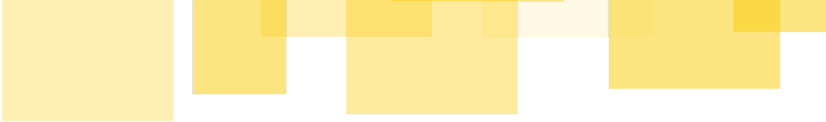
For $t > T$, the formula guarantees that the maximum token amount M has been released:

For $t > T$, $R(t) = M$ (Full release after vesting period)

4. Invariant:

The vesting schedule satisfies the invariant $\forall t, 0 \leq R(t) \leq M$, ensuring that:

- Tokens Released Do Not Exceed the Allocation: At any point during the vesting period, the number of tokens released remains within the total allocation M .
- Proper Enforcement of Vesting Periods: Tokens are released only according to the defined vesting schedule:

- 
- Tokens are locked during the cliff period.
 - Tokens are released linearly after the cliff during the vesting period.
 - The total tokens released do not exceed the allocated M at any time.

Conclusion

The mathematical model and proof demonstrate that the vesting schedule enforces all expected properties:

- Tokens are locked during the cliff period, ensuring no premature release.
- Tokens are released linearly during the vesting period.
- The total tokens released at the end of the schedule are exactly M , preventing over-release or loss.

Thus, the invariant $0 \leq R(t) \leq M$ holds, ensuring the system prevents over-release or premature access to tokens. The vesting schedule is mathematically sound and effectively enforces the expected behavior over time.



Merkle Proof

The Merkle Tree is a cryptographic structure used to validate user claims, such as airdrop allocations. Each claim is represented as a leaf node in the Merkle Tree, encoding user data (e.g., address and claim amount), which is recursively hashed with sibling nodes to construct the tree's root.

To verify a user's claim, a Merkle Proof is provided, which includes the sibling nodes required to reconstruct the root hash. If the reconstructed hash matches the on-chain Merkle Root, the proof is valid, and the claim is authorized.

Mathematical Model

Let:

- $H(x)$ be a cryptographic hash function (e.g., keccak256).
- MerkleRoot be the root hash of the Merkle Tree.
- Leaf_i be the leaf node corresponding to the i -th user, typically $H(\text{address}_i \parallel \text{claim}_i)$.
- $\text{Proof}_i = (h_1, h_2, \dots, h_n)$ be the Merkle proof for the i -th leaf, consisting of the necessary hashes that, when combined, allow the verification of Leaf_i .

The Merkle proof verification function $\text{VerifyProof}(\text{MerkleRoot}, \text{Leaf}_i, \text{Proof}_i)$ computes:

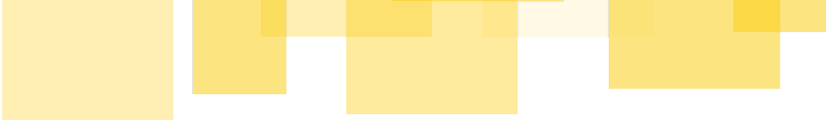
$$H_{\text{computed}} = H(\dots H(H(\text{Leaf}_i \parallel h_1) \parallel h_2) \dots \parallel h_n)$$

where

- \parallel denotes the concatenation of hashes.
- H_{computed} is the root computed from the proof.

The Merkle proof verifies that a given leaf Leaf_i is part of the Merkle Tree with a known root MerkleRoot . The process is as follows:

- Start with the hashed leaf Leaf_i .
- Recursively combine it with the siblings' hashes h_1, h_2, \dots, h_n from the proof, Proof_i , by applying the cryptographic hash function $H(x)$.
- Compare the computed root hash H_{computed} with the known MerkleRoot .



If H_{computed} matches the MerkleRoot, then the proof is valid, confirming Leaf_i 's inclusion in the tree and the user's claim.

Proof

This section proves that the Merkle proof correctly verifies the inclusion of a given user's claim in the Merkle Tree.

1. Correctness of the Leaf Hash:

The leaf node is the hash of the user's address and the claim amount, as an example shown below. The leaf node hash is fixed and can't be altered. It is part of the Merkle proof added in the verification process.

$$\text{Leaf}_i = H(\text{address}_i \parallel \text{claim}_i)$$

2. Proof of Inclusion:

The proof consists of a series of sibling hashes h_1, h_2, \dots, h_n that allow reconstruction of the root hash. Given that the Merkle Tree is binary, each proof step involves a pair of hashes that must be concatenated and hashed together.

The key property of Merkle Trees is that the root hash MerkleRoot can only be generated if the proof is correct and the leaf is part of the tree. The proof ensures the correctness of the leaf's inclusion by ensuring that all intermediate hash steps are valid.

3. Final Computation of Root Hash:

To prove that a leaf is part of the Merkle Tree, we compute the hash recursively as follows:

- First, concatenate the leaf node and the first sibling hash h_1 , then apply the hash function: $H_1 = H(\text{Leaf}_i \parallel h_1)$.
- Then, concatenate H_1 with the next sibling hash h_2 : $H_2 = H(H_1 \parallel h_2)$.
- Continue this process until all sibling hashes are incorporated. Finally, the computed hash should match the original MerkleRoot.

4. If the final computed hash matches the root hash MerkleRoot, the proof is valid, and the user is entitled to the claim.

Conclusion



The mathematical model and proof demonstrate that the Merkle Proof mechanism in the contract guarantees the following:

- Only valid claims (i.e., those included in the Merkle Tree) can be verified.
- The inclusion and correctness of each claim depend solely on the MerkleRoot and Proof_i .
- The computational process ensures that users can independently verify their claims without accessing the entire Merkle Tree.

Thus, the Merkle Proof system securely validates claims.

No Double Claim

The no double claim property ensures that each eligible participant in the airdrop can claim their allocated tokens exactly once. The contract enforces this rule by maintaining a state-tracking mechanism that marks users as having claimed upon their first successful claim. If a user attempts to claim again, the contract must reject the transaction. This prevents unauthorized duplicate claims.

Mathematical Model

Let:

- \mathcal{U} be the set of all eligible users for the airdrop.
- M be the total airdrop token allocation.
- `MerkleRoot` be the root hash of the Merkle Tree used for claim validation.
- `Claimed`[u] be a mapping that records whether a user $u \in \mathcal{U}$ has already claimed their allocation.
- $\text{Leaf}_u = H(\text{address}_u \parallel \text{amount}_u)$ be the Merkle leaf node representing user u 's claim.
- Proof_u be the Merkle proof associated with user u 's claim.
- `claimedAmount` be the total number of tokens successfully claimed by users.

A user u can only claim their allocation once if and only if the following conditions hold:

- The claim is valid under the Merkle Tree, which ensures that the user's claim exists in the Merkle Tree, proving their eligibility.

$$H_{\text{computed}} = H(\text{Leaf}_u \parallel h_1 \parallel h_2 \parallel \dots \parallel h_n) = \text{MerkleRoot}$$

- The user has not already claimed: `Claimed`[u] = `false`. If the value is already set to `true`, the contract must reject any further claims.
- The total claimed tokens never exceed M . This guarantees that the contract never allows more tokens to be claimed than allocated.

$$\sum_{u \in \mathcal{U}} \text{claimedAmount}[u] \leq M$$

Proof

1. Preventing Multiple Claims Per User

The contract records claims using the $\text{Claimed}[u]$ mapping, ensuring that a user u who has successfully claimed once cannot claim again.

- Base Case (Before claim):
 - Assume a user has not yet claimed. By the contract logic, $\text{Claimed}[u] = \text{false}$.
- Claim Execution:
 - If a valid Merkle proof is provided, and $\text{Claimed}[u] = \text{false}$, the contract executes: $\text{Claimed}[u] \leftarrow \text{true}$.
 - The contract transfers amount_u to u .
- Any subsequent claim attempt by u fails because the contract enforces: $\text{require}(\text{Claimed}[u] = \text{false})$, which ensures that once a claim has been successfully processed and marked as claimed, any future attempt to claim again will be rejected by the smart contract, effectively preventing double spending or unauthorized multiple claims.

2. Ensuring Valid Merkle Proofs

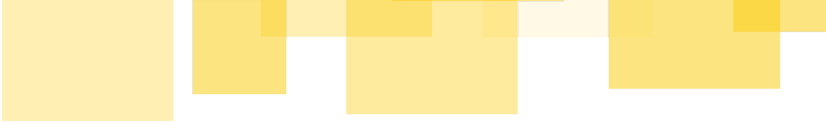
The contract verifies the Merkle proof using the formula below. The transaction is reverted if the proof is invalid, ensuring that only eligible users can claim.

$$H_{\text{computed}} = H(\text{Leaf}_u \parallel h_1 \parallel h_2 \parallel \dots \parallel h_n) = \text{MerkleRoot}$$

3. Preventing Over-Allocation

We ensure that no user can claim beyond the allocated total supply M :

- Each claim is validated against the Merkle proof before execution.
- The contract tracks the total amount of tokens claimed using: $\text{claimedAmount} + \text{amount}_u$
- The contract maintains a total claimed amount invariant:
$$\sum_{u \in \mathcal{U}} \text{claimedAmount}[u] \leq M.$$
- If a claim exceeds the total allocation, it is reverted, preventing any excess claims.



Since each user claims at most once, and the sum of claims does not exceed M , the total allocation is correctly distributed without exceeding limits.

Conclusion

The contract enforces no double claim and correct allocation enforcement property by:

- Storing claimed users in a mapping to prevent reclaims.
- Validating claims against a Merkle Tree, ensuring correctness.
- State updates occurring before external token transfers, ensuring atomicity.
- Maintaining the global allocation constraint to prevent exceeding the total supply.

Since these invariants hold at all times, we formally conclude that no user can claim their tokens more than once, and no user can exceed their entitled allocation, ensuring airdrop integrity and preventing abuse.



Informative Findings

This section highlights observations and recommendations that are worth addressing while not classified as vulnerabilities to improve code quality, maintainability, or performance. These findings may include suggestions for adhering to best practices, documentation improvements, or enhancements to the overall design.



[F01] Temporary Synchronization Delays in FxToken and FxTokenL2 Bridge

Severity: Informative

Recommended Action: Document Prominently

Partially addressed by client

The FxTokenL2 bridge relies on the sequencer's management of minting and burning operations to maintain a 1:1 mapping with FxToken. During periods of high network congestion or transaction volume, temporary synchronization delays can arise due to the time required for finalizing transactions on L1 before they are reflected on L2. These delays are an inherent characteristic of the bridge's design and do not affect the overall integrity or security of the token peg.

The `mint` function in FxTokenL2 demonstrates how tokens are securely minted under the bridge's control:

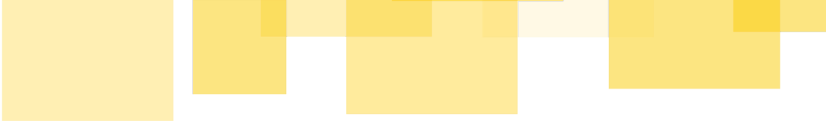
```
function mint(  
    address _to,  
    uint256 _amount  
) external virtual override(IOptimismMintableERC20, ILegacyMintableERC20) onlyBridge {  
    _mint(_to, _amount);  
    emit Mint(_to, _amount);  
}
```

During a period of high transaction volume, a user burns tokens on L1 and expects an equivalent amount of tokens to be minted on L2. Due to L1 congestion, the transaction finalization on L2 can be delayed, and as a result, the corresponding minting on L2 is also delayed. However, the system ensures that the L1-to-L2 transaction will eventually synchronize, maintaining the 1:1 parity between the tokens.

Discussion

The client has acknowledged that the synchronization delays during periods of high network activity are inherent system characteristics and do not pose a security issue. They are aware that the current documentation could benefit from additional clarity regarding this behavior and have committed to updating it.

Recommendation

- 
- Update system documentation to explain the expected behavior during synchronization delays, emphasizing that such delays do not compromise the 1:1 parity or token integrity.
 - While not immediately planned, off-chain monitoring could be considered to track synchronization status between L1 and L2 minting and burning operations, particularly during high network activity. This could help in the timely detection of anomalies.
-

Status

The client has committed to updating documentation to clarify the expected behavior during synchronization delays. Off-chain monitoring tools are not planned for immediate implementation.



[F02] Insufficient Event Emissions for Admin Actions

Severity: Informative

Recommended Action: Fix Code

Addressed by client

While the contracts emit events for critical user actions (e.g., claims in FxAirdrop and TezAirdrop contracts), some admin-level state updates lack event emissions. These updates, such as `setCliff`, `setDuration`, and `setStartTimestamp` in the FxVestingFactory contract, do not currently emit events, which can reduce transparency for off-chain applications and monitoring systems.

Recommendation

No immediate action is necessary, as the current implementation restricts these updates to admin-only functions. However, it is recommended that events be emitted for the admin-level updates in contracts to enhance transparency and monitoring capabilities.

Status

Resolved in PR [#43](#) based on the recommendation from Runtime Verification.



[F03] Centralization Risks in TezAirdrop

Severity: Informative

Recommended Action: Fix Design

Not addressed by client

The TezAirdrop contract relies on a single signer to validate Tezos wallet claims. This design choice enables efficient, autonomous claim verification but introduces a degree of centralization. The primary consideration is that if the signer's private key were compromised, it could allow unauthorized claims.

However, this concern remains theoretical, as the system is designed to function autonomously. Given the need for real-time, automated approvals, requiring a multi-signature setup would not be feasible. The current architecture prioritizes efficiency over decentralization, making this an intentional tradeoff rather than a security flaw.

Recommendation

Implementing periodic key rotation or other key management best practices could enhance security while maintaining the system's autonomous operation.

Status

Acknowledged by the client.