Linea Token and Airdrop Contracts

Date	July 2025
Auditors	George Kobakhidze, Heiko Fisch

1 Executive Summary

This report presents the results of our engagement with the **Linea Team** to review the **Linea Token and Airdrop contracts**.

The review was conducted in the week of **July 21–25, 2025** by **Heiko Fisch** and **George Kobakhidze**.

The audit was conducted over a system of contracts that will facilitate the Linea Token mint, claim, airdrop, and other token functionality. In particular, there were three contracts involved - the token contract on the Ethereum mainnet (L1), the token contract on the Linea rollup (L2), and the airdrop contract that will help distribute the tokens to the eligible recipients on the Linea rollup.

Although the tokens originally get minted on the L1, the primary governance activity using those tokens will happen on the L2. As a result, the L1 token has more functionality relating to its supply and mint mechanics, such as actually being minted by authorized entities as well as having the ability to “sync” its true total supply numbers over to the L2 contract. On the other hand, the L2 token contract has additional governance functionality built in via the `ERC20VotesUpgradeable` inherited contract. Consequently, the communication between the layers is crucial for correct intended governance functionality of the token, and therefore, other offchain systems – such as the Linea Messaging Service – play a critical role in this setup.

No major issues were identified throughout the audit that would threaten the intended logic of the contracts. Due to the nature of airdrops, additional considerations were analyzed, such as simplicity, clarity of the contracts, and potential burden of unwarranted receipt of an airdrop.

After the review of PRs containing fixes for the issues reported in this audit as well as in audits that were ran in parallel by other teams, the final reviewed commit was `91036daf2331610841796772f8fbb2fcc4a9233f` .

The build artifacts provided in the repository’s folder `exported-artifacts` from the final commit also match the artifacts produced by our build runs with the following configurations:

- Compiler: `0.8.30`
- Optimizer runs: `10_000_000`
- L1 contract EVM Version: `Prague`
- L2 contracts EVM Version: `London`

Moreover, on both Ethereum and Linea, the deployed bytecode at address `0xe03f157de67ac4b2a9a949d64d2a3c64ffa1bc55` matches the `deployedBytecode` in the corresponding artifact file in `exported-artifacts` at the final revision `91036daf2331610841796772f8fbb2fcc4a9233f` .

2 Scope

This review focused on the repository and code revision: Consensys/linea-tokens, code revision 44640f0965a5c7465b99769a5d241a9a1cb3a2ef.

The detailed list of files in scope can be found in the Appendix.

3 System Overview

In the following, we briefly describe the main contracts.

3.1 LineaToken

- This is the main token contract, an ERC-20 with mint and burn functionality.
- Minting by the `MINTER_ROLE` , which is managed by the `DEFAULT_ADMIN` . Both roles will be assigned to the security council, according to the Linea team.
- Will be deployed on L1 (Ethereum).
- Tokens can be bridged to L2 (Linea) and also bridged back.
- Upgradeable contract; proxy admin owner will be the same timelock contract as for other upgradeable Linea contracts, according to the Linea team.

3.2 L2LineaToken

- The corresponding token contract on L2.
- Minting directly on L2 is not possible.
- Burning directly on L2 is currently possible but should be removed. (See here.)
- Contract has a `DEFAULT_ADMIN` , which currently has no functionality other than managing the role itself, but will likely be utilized at a later point, after an upgrade. The `DEFAULT_ADMIN` role will be assigned to the security council, according to the Linea team.
- Upgradeable contract; proxy admin owner will be the same timelock contract as for other upgradeable Linea contracts, according to the Linea team.
- Inherits voting functionality from `ERC20VotesUpgradeable` .

3.3 MessageServiceBase

- `L2LineaToken` inherits from this contract.
- Taken from a different Linea codebase, with minimal changes only, to avoid code duplication.

3.4 TokenAirdrop

- Non-upgradeable airdrop contract.
- Will be deployed on L2.
- Funds for the airdrop have to be bridged to L2 and then transferred to this contract.
- Has a claim window that is set in the constructor.
- After that, the owner can withdraw all remaining funds.
- Allocation amounts are calculated based on (up to) three SBT ERC-20 contracts, whose balances have to be set accordingly.

4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under review. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the review team.

4.1 Actors

The relevant actors are listed below with their respective abilities:

- Proxy Admin Owner. The token contracts on both L1 and L2 are upgradeable. The Proxy Admin Owner is the address that can perform such an upgrade. According to the Linea team, it will be set to the same timelock contract that is also the Proxy Admin Owner of other upgradeable Linea contracts.
- Default Admin. There is a Default Admin assigned to both L1 and L2 contracts of the Linea token. This admin manages all roles in the system, including the Default Admin role itself. According to the Linea team, this role will be assigned to the security council.
- Minter Role. The Minter is explicitly assigned on the L1 token contract of the Linea token, and they are able to arbitrarily mint tokens to any recipient. It is important to note that they are not able to burn tokens from anyone, only mint. According to the Linea team, this role will also be assigned to the security council.
- Linea Message Service. Though not an entity that is explicitly assigned any roles in the contracts, the Linea Message Service plays a critical role in the system, which facilitates messages between L1 and L2. Specifically, the two main mechanics are bridging between the layers and coordinating the total token supply from L1 to L2.
- Owner of the Airdrop Contract. The Owner can withdraw all unclaimed tokens when the airdrop window has closed. They can also transfer the Owner role to someone else.
- Deployers. The deployers of the contracts control the initial (and sometimes unchangeable) values for many system parameters.
- Users. Regular users of the system can do well-known token operations such as transfers, approvals, and burning of their own tokens. They can also bridge tokens between L1 and L2, and they can vote on L2.

4.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this review, we established the following trust model:

- Proxy Admin Owner. The ability to upgrade the token contract gives the Proxy Admin Owner complete control over the code as well as the data stored in the contract. If the Proxy Admin Owner is itself a contract (such as a timelock), the restrictions of this contract can limit the power (e.g., upgrade is only executed after a certain time has passed).
- Default Admin. As addresses with the Default Admin role can manage all roles, they can grant and revoke the Default Admin and the Minter role to/from other addresses (including themself). They are trusted to manage these roles responsibly.
- Minter Role. Though the Minter can not modify any logic explicitly, they are able to create Linea tokens. Therefore, they can financially impact the Linea token by inflating the supply and selling the tokens on the open market if the Minter address is compromised. Similarly, an inflated token supply can be used to sway any ongoing governance votes on the L2, thereby granting some governance power to the Minter indirectly. As such, the Minter is trusted not to abuse such supply mechanics.
- Linea Message Service. The entity responsible for controlling the Linea Message Service is not able to modify or influence the messages being sent. However, they are able to stop any messages from coming if they choose not to execute them, effectively censoring them. In our case, such messages would be bridging the tokens between the layers and communicating the supply numbers – both critical events for effective governance of the Linea token system. Therefore, the Message Service is trusted not to censor any message.
- Owner of the Airdrop Contract. Technically, no special trust is required in the Owner, but there is the assumption that the Owner or someone else will fund the airdrop contract sufficiently for all allocations to be claimable.
- Deployers. No trust is required for the deployers, as the deployment parameters can be verified on-chain.
- Users. Users are only in control of their own balances.

4.3 Additional Notes

- Dependencies. The Linea Token contracts make extensive use of open source libraries, specifically OpenZeppelin contracts and their upgradeable counterparts. In order to effectively secure the system and assess its risks, it is critical that versions of such dependencies are locked and recorded. Due to its specific setup, this wasn't explicitly mentioned in the repository of the system. However, by the end of the audit, the OpenZeppelin contracts' version has been fixed at 5.4.0.

5 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

5.1 Burning on L2 Is Not Reflected in the Total Supply Medium ✓ Fixed

Resolution

This has been fixed in PR 11 (last commit on the branch: b9aad54).

Description

Minting tokens is only possible on the L1, not on the L2 (except *technically*, as L2 counterpart for the tokens locked in the L1 bridge, but not semantically). Hence, the `totalSupply` on the L1 token reflects (or is *supposed* to reflect) the entire supply across L1 and L2, while `totalSupply` on the L2 token only reflects the supply on L2.

However, as `L2LineaToken` inherits from `ERC20BurnableUpgradeable`, burning (as opposed to minting) *is* possible on L2, but this will not be reflected in the `totalSupply` of the L1 token. Instead, `totalSupply` on L1 will overestimate the real total supply by the amount of tokens that have been burnt on L2. These could also never be re-minted as their L1 counterparts are stuck on the L1 bridge.

Hence, in order for `totalSupply` on L1 to accurately reflect the total supply across L1 and L2, the possibility to burn on the L2 should be removed (except, of course, the *technical* burning of tokens that are bridged back to the L1).

Recommendation

We recommend removing the inheritance from `ERC20BurnableUpgradeable` in `L2LineaToken`. Burning directly on L2 won't be possible then; instead, tokens that are currently on L2 and should be burned have to be bridged back to L1 first and can then be burned there.

Remark

This was brought up by thedarkjester from the Linea team in a call with the client at the beginning of the engagement.

5.2 `L2LineaToken`: `_msgSender()` Should Be Replaced With `msg.sender` Minor ✓ Fixed

Resolution
This has been fixed in PR 15 (last commit on the branch: 469fe87).

Description

There are three occurrences of `_msgSender()` in `L2LineaToken`, all in the `mint` and `burn` function:

src/L2/L2LineaToken.sol:L70-L94

```
/**
 * @notice Mints the Linea token.
 * @dev NB: Only the L2 token bridge can call this function.
 * @param _account Account being minted for.
 * @param _amount The amount being minted for the account.
 */
function mint(address _account, uint256 _amount) external {
  require(_msgSender() == lineaCanonicalTokenBridge, CallerIsNotTokenBridge());

  _mint(_account, _amount);
}

/**
 * @notice Burns the Linea token.
 * @dev NB: Only the L2 token bridge can call this function.
 * @dev Approval for the burn amount must be provided before this is invoked.
 * @param _account Account being burned for.
 * @param _value The amount being burned for the account.
 */
function burn(address _account, uint256 _value) external {
  require(_msgSender() == lineaCanonicalTokenBridge, CallerIsNotTokenBridge());

  _spendAllowance(_account, _msgSender(), _value);
  _burn(_account, _value);
}
```

While in the current state of the codebase, `_msgSender()` returns `msg.sender` – which means there is no functional difference between the two – the function *could* be overridden to return a different value. In `mint` and `burn`, however, it seems certain that we'd always want to know if the call really comes from the canonical token bridge, even if some meta transaction functionality should ever be introduced.

Moreover, `L2LineaToken` inherits from `MessageServiceBase` and uses its `onlyMessagingService` modifier, which has related functionality: checking whether the call comes from the message service. And this modifier uses `msg.sender`:

src/MessageServiceBase.sol:L48-L53

```
modifier onlyMessagingService() {
  if (msg.sender != address(messageService)) {
    revert CallerIsNotMessageService();
  }
  _;
}
```

Recommendation

We recommend replacing `_msgSender()` with `msg.sender` in `L2LineaToken.mint` and `L2LineaToken.burn`.

5.3 Can Claim for Other Users Without Their Consent Minor ✓ Fixed

Resolution
This has been fixed in PR 14 (last commit on the branch: 40f5d75).

Description

The `TokenAirdrop` contract allows users to claim tokens based on their available calculation. This is done by executing the claim function with an address to claim for:

src/airdrops/TokenAirdrop.sol:L120-L138

```
/**
 * @notice Claims tokens for an account while the claiming is active.
 * @dev Claiming sets claim status pre-transferring avoiding reentry, and all multiplier tokens are soulbound avoiding
 * transfer manipulation.
 * @param _account Account being claimed for.
 */
function claim(address _account) external {
  require(block.timestamp < CLAIM_END, ClaimFinished());
  require(!hasClaimed[_account], AlreadyClaimed());

  uint256 tokenAmount = calculateAllocation(_account);

  require(tokenAmount != 0, ClaimAmountIsZero());

  hasClaimed[_account] = true;
  emit Claimed(_account, tokenAmount);

  TOKEN.safeTransfer(_account, tokenAmount);
}
```

As it can be seen above, anyone can initiate the claim for any `account`, and the tokens will indeed be transferred to the appropriate recipient. However, this is done without any explicit consent of the account. This may have two impacts.

First, this has personal implications for the `account` owner, such as tax implications. In many jurisdictions a person is taxed upon receipt of a valuable asset, whether they wanted it or not. While anyone indeed can send anything to anyone, such as stablecoins to a known ENS, in this case the user who initiated the `claim()` call does not actually spend any of their own funds. They simply claim the airdrop for the `account` and trigger a potential taxable event, which the recipient potentially wanted to delay until the next tax year, for example.

Second, not all eligible accounts may be owned by their owners securely. For example, the private keys could have been lost. In that case, claiming and sending tokens to that account could either effectively burn these tokens. Instead, these tokens could have been used by the foundation or for other needs after `CLAIM_END`.

To sum it up, the explicit consent of the owners of eligible `account` addresses would go a long way in such an airdrop contract. This is something that has also been considered in other large airdrops in the cryptocurrency industry.

Recommendation

Consider requiring explicit consent of the `account` address owners, such as providing a signed message from the `account` or initiating the claim transaction from the `account` address and claiming on `msg.sender`.

5.4 Frontrunning L1 `LineaToken` Supply Syncs Minor Acknowledged

Resolution
The Linea team acknowledged this as low risk due to how they expect their operations to run. In particular, any token mints on the L1 are not intended to be followed by a supply sync call immediately after in the same transaction or block, removing the frontrunning risk.

Description

The `LineaToken` contract consists of two parts - one on the L1 and one on the L2. The true token lives on the L1, and, therefore, its true total supply is governed by its L1 `totalSupply()` function. Similarly, all mints and burns that actually affect the true number of tokens in circulation also happen on the L1 contract.

However, the L2 token is what will primarily help with governance use cases, such as voting, as can be seen by the choice to make it inherit from the `ERC20VotesUpgradeable` contract, whereas the L1 token does not. As a result, the L2 token requires knowing the true total supply of the token, i.e. the one from L1, to make meaningful governance proposals and execute upon them as that would require quorum from all possible tokens.

In order to do that, the L1 token contract has a function that initiates the sync:

src/L1/LineaToken.sol:L72-L86

```
  /**
   * @notice Synchronizes the total supply of the L1 token to the L2 token.
   * @dev This function sends a message to the L2 token contract to sync the total supply.
   * @dev NB: This function is permissionless on purpose, allowing anyone to trigger the sync.
   * @dev This function can only be called after the L2 token address has been set.
   */
  function syncTotalSupplyToL2() external {
    uint256 totalSupply = totalSupply();

    /// @dev Fee is set to 0 and should be automatically claimed on Linea.
    IMessageService(l1MessageService).sendMessage(l2TokenAddress, 0, abi.encodeCall(IL2LineaToken.syncTotalSupplyFromL1, (block.timestamp, totalSupply)));

    emit L1TotalSupplySyncStarted(block.timestamp, totalSupply);
  }
}
```

This simply collects the current token supply, `block.timestamp` and calls the Linea `l1MessageService` contract responsible for transferring messages. On the L2 side, the token contract expects its the message service contract to transfer this message and only accepts it if the original caller is the L1 token contract:

src/L2/L2LineaToken.sol:L96-L111

```
/**
 * @notice Synchronizes the total supply of the L1 Linea token from L1 Ethereum.
 * @dev NB: This function can only be called by the Linea Message Service.
 * @dev NB: This function must have originated from the Linea token on L1 Ethereum.
 * @param _l1LineaTokenTotalSupplySyncTime The L1 block.timestamp when the Linea token on L1 total supply was
 * computed.
 * @param _l1LineaTokenSupply The total supply of the L1 Linea token.
 */
function syncTotalSupplyFromL1(uint256 _l1LineaTokenTotalSupplySyncTime, uint256 _l1LineaTokenSupply) external onlyMessagingService onlyAuthorizedRemoteSender
  require(l1LineaTokenTotalSupplySyncTime < _l1LineaTokenTotalSupplySyncTime, LastSyncMoreRecent());

  l1LineaTokenSupply = _l1LineaTokenSupply;
  l1LineaTokenTotalSupplySyncTime = _l1LineaTokenTotalSupplySyncTime;

  emit L1LineaTokenTotalSupplySynced(_l1LineaTokenTotalSupplySyncTime, _l1LineaTokenSupply);
}
```

The interesting part is the check `require(l1LineaTokenTotalSupplySyncTime < _l1LineaTokenTotalSupplySyncTime, LastSyncMoreRecent());`. While it does achieve its primary objective of not allowing older supply syncs to overwrite its state, it also prevents several updates from the same block.

In very specific circumstances, this could open up an attack vector. For example, if Linea L1 token mints would happen in the same block as the supply syncs, for example to make sure the L2 side has current total supply numbers, an attacker could listen to these transactions in the mempool, frontrun the mint transaction with their own sync call, and send the old supply numbers over to the L2. Since the updated supply sync call would be in the same block, both of those transactions would have the same `block.timestamp`. As a result, once they reach the L2, the first sync with old numbers would successfully execute, whereas the second one would not as it wouldn't pass the `require()` statement.

The impact would simply be that another sync would have to be initiated from the L1 in the next block or later. Thankfully, the `syncTotalSupplyToL2()` function is externally accessible with no special permissions, so anyone could call it again. However, this would still make a brief window appear where the L1 token supply and its corresponding `l1LineaTokenSupply` value are not synchronized. Depending on the case, such as if anyone even notices this discrepancy, it could impact governance votes and proposals for longer than preferable.

Recommendation

Consider token supply and syncs operations carefully, such as ensuring the syncs happen correctly after any supply changes on the L1 side.

5.5 `LineaToken` and `L2LineaToken`: Missing Sanity Checks During Initialization Minor ✓ Fixed

Resolution
This has been fixed in PR 10 (last commit on the branch: 86605a9).

Description

The `initialize` functions in `LineaToken` and `L2LineaToken` both take several `address` arguments as well as the name and symbol of the token. While, throughout the codebase, every address argument given in an `initialize` function or a constructor is diligently verified to be non-zero, the token name and symbol are not verified to be non-empty strings – although that would just as well constitute a deployment mistake and justify reverting.

Recommendation

For completeness and consistency, we recommend checking in both `LineaToken.initialize` and in `L2LineaToken.initialize` that the string arguments `_tokenName` and `_tokenSymbol` are non-empty.

5.6 `TokenAirdrop`: Questionable Use of `selfdestruct` Minor ✓ Fixed

Resolution

This has been fixed in PR 12 (last commit on the branch: 4e0391d).

Description

When the claim window for an airdrop has closed, the owner of the contract can withdraw all unclaimed tokens. At the end of the `withdraw` function, `selfdestruct` is called:

src/airdrops/TokenAirdrop.sol:L140-L153

```
/**
 * @notice Owner withdraws unclaimed tokens from the contract post claim end and self destructs.
 * @dev The selfdestruct only applies for pre EIP-6780 forks.
 */
function withdraw() external onlyOwner {
  require(CLAIM_END <= block.timestamp, ClaimNotFinished());

  uint256 balance = TOKEN.balanceOf(address(this));

  emit TokenBalanceWithdrawn(msg.sender, balance);
  TOKEN.safeTransfer(msg.sender, balance);

  selfdestruct(payable(msg.sender));
}
```

Since the contract has no payable function, this is not about collecting ETH stored in the contract. Instead, Linea runs on the London EVM, so `selfdestruct` actually removes the code, which is not true for chains that have included EIP-6780.

While this would have once counted as exemplary smart contract programming, these days using `selfdestruct` is strongly discouraged. The Solidity documentation is very explicit in this regard:

> Also, note that the selfdestruct opcode has been deprecated in Solidity version 0.8.18, as recommended by EIP-6049. The deprecation is still in effect and the compiler will still emit warnings on its use. Any use in newly deployed contracts is strongly discouraged even if the new behavior is taken into account. Future changes to the EVM might further reduce the functionality of the opcode.

Even though the point of using `selfdestruct` in this situation on Linea (or a pre EIP-6780 chain in general) can be seen, `selfdestruct` is widely regarded as anti-pattern these days and even frowned upon, so that we side with the Solidity documentation and recommend not using it anymore in newly deployed contracts.

Recommendation

Consider removing the `selfdestruct(payable(msg.sender));`.

5.7 `LineaToken`, `L2LineaToken`, and `TokenAirdrop`: Consider Emitting Events During Initialization ✓ Fixed

Resolution

The following events have been added in PR 10 (last commit on the branch: 86605a9):

- `L2TokenAddressSet` is emitted in `LineaToken.initialize`
- `L1TokenAddressSet` is emitted in `L2LineaToken.initialize`

Additionally, new events were introduced in PR 16 (last commit on the branch: 3a9695a):

- `TokenMetadataSet` in `LineaToken.initialize` and `L2LineaToken.initialize`
- `L1MessageServiceSet` in `LineaToken.initialize`
- `L2MessageServiceSet` and `LineaCanonicalTokenBridgeSet` in `L2LineaToken.initialize`
- `AirdropConfigured` in `TokenAirdrop.constructor`

Description

The `initialize` functions in `LineaToken` and `L2LineaToken` as well as the constructor in `TokenAirdrop` each set a bunch of (unchangeable) state variables to the values given as arguments. With a few exceptions, no events are emitted with the values these state variables were set to. While they are all `public` – which means a getter function to query their value is automatically generated – it might still be nice and consistent to make some or all of these values discoverable via events.

This affects:

- The token name and symbol as well as `l1MessageService` and `l2TokenAddress` in `LineaToken`. (Note that `_grantRole` already emits an event.)
- The token name and symbol as well as `messageService` and `lineaCanonicalTokenBridge` in `L2LineaToken`. (Note that `_grantRole` and `_setRemoteSender` already emit events.)
- `TOKEN`, `PRIMARY_FACTOR_ADDRESS`, `PRIMARY_CONDITIONAL_MULTIPLIER_ADDRESS`, `SECONDARY_FACTOR_ADDRESS`, and `CLAIM_END` in `TokenAirdrop`.

Recommendation

Consider adding events as indicated above. Note that the event `L2TokenAddressSet` is defined in `ILineaToken` but never emitted. It could be used in this context; if it's not, it should be removed.

5.8 `LineaToken`, `L2LineaToken`, and `MessageServiceBase`: Consider Using Namespaced Storage Acknowledged

Description

The L1 and L2 token contracts are upgradeable and use openzeppelin-contracts-upgradeable in v5, which utilizes namespaced storage. The token contracts themselves, however, use regular state variables.

While namespaced storage is a bit more verbose and harder to read, it significantly reduces the risk of storage-layout-related bugs in contract upgrades, so OZ generally recommends using it in self-implemented child contracts too, especially if these contracts are (or might become) base contracts for other contracts.

Recommendation

Consider using namespaced storage in `LineaToken`, `L2LineaToken`, and `MessageServiceBase`, as this would bring the advantages discussed above. On the other hand, currently there are no contracts which inherit from `LineaToken` and `L2LineaToken` (which might or might not change in the future), and – while `L2LineaToken` itself inherits from `MessageServiceBase` – it was an explicit intention to make as few changes as possible in the latter contract, as it has already been audited before. In summary, while we think that namespaced storage would be preferable in principle, we believe in this particular case, using a traditional storage layout is a defendable design choice too.

5.9 `TokenAirdrop`: Minor Code Simplification Opportunity ✓ Fixed

Resolution
This has been fixed in PR 10 (last commit on the branch: 86605a9).

Description

The `TokenAirdrop` contract has (among others) the immutable state variables `PRIMARY_FACTOR_ADDRESS` and `PRIMARY_CONDITIONAL_MULTIPLIER_ADDRESS`. Each of these stores either an address of an ERC-20 contract or the zero address, and – if not zero – the balances in these contracts contribute to the calculation of the airdrop amounts:

src/airdrops/TokenAirdrop.sol:L110-L113

```
function calculateAllocation(address _account) public view returns (uint256 tokenAllocation) {
  if (address(PRIMARY_FACTOR_ADDRESS) != address(0) && address(PRIMARY_CONDITIONAL_MULTIPLIER_ADDRESS) != address(0)) {
    tokenAllocation = (PRIMARY_FACTOR_ADDRESS.balanceOf(_account) * PRIMARY_CONDITIONAL_MULTIPLIER_ADDRESS.balanceOf(_account)) / DENOMINATOR;
  }
```

However, the contract's constructor makes sure that either both addresses are zero or none of them:

src/airdrops/TokenAirdrop.sol:L92-L94

```
require(_primaryFactorAddress != address(0) || _primaryConditionalMultiplierAddress == address(0), ZeroAddressNotAllowed());

require(_primaryFactorAddress == address(0) || _primaryConditionalMultiplierAddress != address(0), ZeroAddressNotAllowed());
```

Hence, the condition

```
address(PRIMARY_FACTOR_ADDRESS) != address(0) && address(PRIMARY_CONDITIONAL_MULTIPLIER_ADDRESS) != address(0)
```

in the first code excerpt above can actually be simplified to:

```
address(PRIMARY_FACTOR_ADDRESS) != address(0)
```

Recommendation

Consider simplifying the condition in the `if` as indicated above. Maybe add a comment that the constructor checks guarantee that either both state variables are zero or none of them.

Appendix 1 - Files in Scope

This review covered the following files.

Initial version at revision 44640f0965a5c7465b99769a5d241a9a1cb3a2ef:

File	SHA-1 hash
src/L1/LineaToken.sol	`1e65975280fc6c7b354b95f2cd8afa28622df43a`
src/L1/interfaces/ILineaToken.sol	`ebd435b63696df7088d80844892e428077c975dd`
src/L2/L2LineaToken.sol	`77359f138e309486d7b867dd89815a783c280199`
src/L2/interfaces/IL2LineaToken.sol	`9ffff46fc5d5ad69a1c957ce6601b5f8ea5517b6`
src/MessageServiceBase.sol	`ef9f57d3f7ee879b70ceed8f7b72f7528fa41106`
src/airdrops/TokenAirdrop.sol	`1a86a21606f6e0a6a6da5cb6d82e473d1658ee5f`
src/interfaces/IGenericErrors.sol	`6677ac4576d10fd500b487d2f19613ea373cb79c`
src/interfaces/IMessageService.sol	`7bb358e04f57317bba68d0c60451a91e3bdbbb05`

Final version at revision 91036daf2331610841796772f8fbb2fcc4a9233f:

File Name	SHA-1 Hash
src/L1/LineaToken.sol	`d8a7b3cb47faef6ab4646271a0a4dd1a5817c35d`
src/L1/interfaces/ILineaToken.sol	`96546c3b426f57496db6850fdf73fea10a7933f7`
src/L2/L2LineaToken.sol	`ce2599b6d8200d914eb466c73d2795ae0cf438fb`
src/L2/interfaces/IL2LineaToken.sol	`67a16b0362153d9ededcc70a0225341adf0bc569`
src/MessageServiceBase.sol	`ef9f57d3f7ee879b70ceed8f7b72f7528fa41106`
src/airdrops/TokenAirdrop.sol	`c2cf4a691ad6eb1d6e83803069e5939769653bf1`
src/interfaces/IGenericErrors.sol	`6677ac4576d10fd500b487d2f19613ea373cb79c`
src/interfaces/IMessageService.sol	`7bb358e04f57317bba68d0c60451a91e3bdbbb05`

Appendix 2 - Disclosure

Consensys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of

making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.