CANTINA

Horizen migration

Security Review

Cantina Managed review by:

M4rio.eth, Lead Security Researcher
Sujith Somraaj, Security Researcher

May 29, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Horizen empowers builders with flexible privacy and proof-driven trust, allowing secure and innovative decentralized applications to thrive.

From Apr 30th to May 1st the Cantina team conducted a review of horizen-migration on commit hash f4156fee, along with a set of new claim methods and dump script (horizen-migration and horizen-migration-check):

- Dump Scripts:
  - `get_all_forger_stakes.py`
  - `setup_eon2_json.py`
  - `zend_to_horizen.py`
- Two Claim Methods added to `ZendBackupVault.sol` (Tag `1.3-RC1`)

The team identified a total of **16** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	0	1
Low Risk	6	4	2
Gas Optimizations	1	1	0
Informational	8	7	1
Total	**16**	**12**	**4**

3 Findings

3.1 Medium Risk

3.1.1 Excessive admin controls in `LinearTokenVesting` could break vesting invariants

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `LinearTokenVesting.sol` contract grants excessive powers to the immutable admin address, which can be exploited to:

- Instantly unlock all vested tokens by manipulating vesting parameters.
- Permanently deny beneficiaries access to their tokens by continuously resetting parameters (Denial of Service).

Description: The `changeVestingParams()` function gives the **admin** the ability to reset the vesting schedule after it has already started completely.

This function:

- Allows modification of `timeBetweenClaims` and `intervalsToClaim` with caller-supplied values.
- Resets the vesting clock (startTimestamp = block.timestamp).
- Resets claim progress (intervalsAlreadyClaimed = 0).

```
function changeVestingParams(uint256 newTimeBetweenClaims, uint256 newNumberOfIntervalsToClaim) public isAdmin {
    if (intervalsAlreadyClaimed == intervalsToClaim) revert UnauthorizedOperation();
    uint256 oldTimeBetweenClaims = timeBetweenClaims;
    uint256 oldNumberOfIntervalsToClaim = intervalsToClaim;
    _setVestingParams(newTimeBetweenClaims, newNumberOfIntervalsToClaim);

    // if startVesting was already called, startTimestamp, amountForEachClaim and intervalsAlreadyClaimed need
    ↪  to be reset
    if (startTimestamp != 0){
        uint256 totalToVest = token.balanceOf(address(this));
        amountForEachClaim = totalToVest / intervalsToClaim;
        startTimestamp = block.timestamp;
        intervalsAlreadyClaimed = 0;
    }
    emit ChangedVestingParams(newTimeBetweenClaims, newNumberOfIntervalsToClaim, oldTimeBetweenClaims,
    ↪  oldNumberOfIntervalsToClaim);
}
```

- **Vulnerability 1: Instant Token Release:** The admin can unlock 100% of the remaining balance instantly (`changeVestingParams(1, 1)` → wait 1 second → `claim()`). However, in the factory the fields that denote the intervals and time in between are "immutable" values (30 days 48 intervals) which might mean the vesting should be immutable.
- **Vulnerability 2: Denial of Service (DoS):** Conversely, a malicious admin can permanently prevent token claims by repeatedly calling `changeVestingParams()` just before the claim period ends. Since each parameter change resets the vesting clock and claim progress, a malicious or compromised admin can effectively hold the tokens hostage indefinitely by continuously resetting the vesting schedule, creating a permanent DoS condition.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console } from "forge-std/Test.sol";
import {ZenToken} from "src/ZenToken.sol";
import {LinearTokenVesting} from "src/LinearTokenVesting.sol";

contract AuditTest is Test {
    ZenToken public zenToken;

    address eonBackup = makeAddr("eonBackup");
    address zenBackup = makeAddr("zenBackup");
```

```
    address admin = makeAddr("admin");
    address beneficiary = makeAddr("beneficiary");

    address foundation;
    address dao;

    function setUp() public {
        foundation = address(new LinearTokenVesting(admin, beneficiary, 1 days, 10));
        dao = address(new LinearTokenVesting(admin, beneficiary, 1 days, 10));

        zenToken = new ZenToken("Zen", "ZEN", eonBackup, zenBackup, foundation, dao);

        LinearTokenVesting(dao).setERC20(address(zenToken));
        LinearTokenVesting(foundation).setERC20(address(zenToken));
    }

    function test_dos() public {
        vm.startPrank(eonBackup);
        zenToken.notifyMintingDone();

        vm.startPrank(zenBackup);
        zenToken.notifyMintingDone();

       console.log(LinearTokenVesting(dao).amountForEachClaim() * 33);
       console.log(zenToken.balanceOf(dao));

       vm.warp(block.timestamp + 1 days);

       vm.startPrank(admin);
       LinearTokenVesting(foundation).changeVestingParams(1 days, 100);

       vm.warp(block.timestamp + 1);
       LinearTokenVesting(foundation).claim();
    }

    function test_instant_claim() public {
         vm.startPrank(eonBackup);
        zenToken.notifyMintingDone();

        vm.startPrank(zenBackup);
        zenToken.notifyMintingDone();

        vm.startPrank(admin);
       LinearTokenVesting(foundation).changeVestingParams(1, 1);

        vm.warp(block.timestamp + 1);
        LinearTokenVesting(foundation).claim();
    }
}
```

Recommendation: Consider either making the vesting immutable which means it can not be modified or consider setting some boundaries like: the new vesting period can not be lower than the remaining time.

Horizen: Acknowledged. While the vesting contract includes admin-level rights, control is not centralized. Permissions are gated behind a multi-signature wallet managed by several parties, ensuring no single actor can alter vesting parameters or disrupt token distribution. This structure supports DAO decisions and upholds collective, transparent governance.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 Faulty `batchInsert` calls in `EONBackupVault` and `ZendBackupVault` could lead to irrecoverable state

Severity: Low Risk

Context: EONBackupVault.sol#L63, ZendBackupVault.sol#L94

Description: The `EONBackupVault.sol` and `ZendBackupVault.sol` contracts are designed to migrate balances from the old EON and ZEND chains to the new ZEN token contract on Base. The migration process

involves setting a cumulative hash checkpoint and then inserting batches of address-value pairs that contribute to a running hash.

Although the `batchInsert()` function is called by only the owner, there are still a few operational risks that need to be addressed.

1. **Duplicate Address Entries:** The `batchInsert()` function has no mechanism to detect or properly handle duplicate addresses across different batches. It silently overwrites previous balances with new values, which may result in users receiving incorrect token amounts (typically less than they are entitled to).
2. **Exceeding Total Supply Cap:** `ZenToken` has a hard cap of 21,000,000 tokens, but `EONBackupVault` has no validation to ensure the sum of inserted balances stays below this cap.
3. **Corruption during insertion:**Any manual error during data insertion could result in an irreparable state, where the entire set of contracts must be redeployed.

Recommendation: As new minters cannot be added after deployment, ensure the entire process is complete and good off-chain before deployment. If possible, consider adding guardrails to avoid duplicates and supply cap exceeding issues in the on-chain `batchInsert()` function.

Horizen: Acknowledged the concern on data loading. To address this, we will manage the loading process with a automated script, which has undergone thorough testing, and will be employed to mitigate risks and ensure data integrity.

Cantina Managed: Acknowledged.

3.2.2 Possible replay attack risk in `ZendBackupVault`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `createMessageHash()` function in `ZendBackupVault` for claim signature verification lacks essential binding elements, creating vulnerability to replay attacks. The current implementation only includes:

```
// ZendBackupVault
string memory strMessageToSign = string(abi.encodePacked(message_prefix, asString));
bytes32 messageHash = VerificationLibrary.createMessageHash(strMessageToSign);

/// Verification Library
function createMessageHash(string memory message) internal pure returns(bytes32) {
  bytes memory messageToSignBytes = bytes(message);
  bytes memory mmb2 = abi.encodePacked(uint8(MESSAGE_MAGIC_BYTES.length), MESSAGE_MAGIC_BYTES);
  bytes memory mts2 = abi.encodePacked(uint8(messageToSignBytes.length), messageToSignBytes);

  // array concatenation
  bytes memory combinedMessage = abi.encodePacked(mmb2, mts2);

  // Double SHA-256 hashing
  return sha256(abi.encodePacked(sha256(combinedMessage)));
}
```

The `createMessageHash()` function processes only the message content without including:

- Chain ID (EIP 155).
- Contract Address.

Hard Fork Replay Attack: Signatures valid on the original chain can be replayed on the forked chain, which has the potential for double-spending migrated balances.

Replaying testnet transactions on mainnet: The entire migration process is staged on a testnet before deployment on Base. Hence, those test transactions could be replayed on the mainnet (and if the `destAddress` is intended to be different, those funds could be at risk).

Recommendation: To avoid replay attacks, consider encoding the chain ID and address of the ZendBackupVault contract in the message hash alongside fork detection.

Horizen: Acknowledged. To avoid test transactions replay we will perform each test run with a claim message prefix different to the one used in production.

Cantina Managed: Acknowledged.

3.2.3 Storing unordered eon vault data in `zend_to_horizen` script

Severity: Low Risk

Context: zend_to_horizen.py#L137.

Description: The script creates a sorted version of the EON vault accounts (`sorted_eon_vault_accounts`) but incorrectly saves the original unsorted dictionary (`eon_vault_results`) to the output file.

```
sorted_eon_vault_accounts = collections.OrderedDict(sorted(eon_vault_results.items()))

with open(eon_vault_result_file_name, "w") as jsonFile:
    json.dump(eon_vault_results, jsonFile, indent=4)  # ← Bug: saves unsorted data
```

Recommendation: Consider storing `sorted_eon_vault_accounts` as follows:

```
sorted_eon_vault_accounts = collections.OrderedDict(sorted(eon_vault_results.items()))

with open(eon_vault_result_file_name, "w") as jsonFile:
    json.dump(sorted_eon_vault_accounts, jsonFile, indent=4)  # ← Fix: saves sorted data
```

Horizen: Fixed in commit e89c80a7.

Cantina Managed: Fix verified.

3.2.4 Missing balance assertion in `zend_to_horizen` script

Severity: Low Risk

Context: zend_to_horizen.py#L127.

Description: The `zend_to_horizen` script lacks assertions to ensure all balances are appropriately accounted for during migration. There is no verification that the sum of migrated balances equals the total input balance:

```
**Should be added after balance totaling is complete:** total_balance = total_balance_to_zend_vault +
↪  total_balance_to_eon_vault + total_balance_not_migrated
assert total_balance_to_zend_vault + total_balance_to_eon_vault + total_balance_not_migrated == total_balance,
↪  "Balance totals mismatch"
```

The lack of validation could lead to:

- Silent balance loss could go undetected.
- No early warning if migration logic fails.

Recommendation: Add an assertion to validate the balance throughout the migration process.

Horizen: Fixed in commit 120a21a3.

Cantina Managed: Fix verified.

3.2.5 Missing duplicates check could result in incorrect data being processed and generated

Severity: Low Risk

Context: zend_to_horizen.py#L77, get_all_forger_stakes.py, setup_eon2_json.py#L43

Description: The `get_all_forger_stakes.py`, `setup_eon2_json.py` and `zend_to_horizen.py` are scripts used to restore EON accounts and migrating Zend balances inside the Horizen state:

The data from EON are:
- the account data, dumped with "zen_dump" rpc command.
- the list of delegators with the amount of their stakes, retrieved using `get_all_forger_stakes.py` script. The stake amounts are added to the delegator account balance.

These accounts will be directly restored in the Zen ERC20 smart contract, with the same balances they had in EON, using the EonBackupVault smart contract.
The data from Zend are a list of Zend addresses with their balance. These accounts cannot be directly restored in the Zen ERC20 smart contract, because the destination address cannot be automatically determined. So the owners of these accounts that want to import their balances in Horizen 2 will need to execute a claim procedure, specifying a Horizen account where their funds will be sent. This claim procedure will be executed using ZenBackupVault smart contract.
Note: There can be cases where some Zend accounts cannot be restored using the claim procedure. In that case, the Ethereum address where their funds will be restored will be provided directly off-chain by the owners, using a json file where the Zend accounts are mapped to Ethereum addresses. These accounts will then be restored using the EonBackupVault smart contract, as if they were Eon Accounts.

The workflow of these scripts is as follows:

- Execute the dump on Zend using the dumper application.
- Convert the Zend dump using the `zend_to_horizen.py` script, together with the Zend-Ethereum address mapping file if provided, and then retrieve the output files: one for the addresses to be restored using the `ZenBackupVault` smart contract and one for the `EonBackupVault` smart contract (e.g., `zend_vault_accounts.json` and `eon_vault_accounts.json`).
- Call the `zen_dump` RPC method on EON at a certain block height and retrieve the resulting file (e.g., `eon_dump.json`).
- Execute the `get_all_forger_stakes` script at the same block height used with the `zen_dump` RPC and retrieve the resulting file (e.g., `eon_stakes.json`).
- Execute the `setup_eon2_json` script using as input the EON dump file, the EON stakes file, and the file with the Zend accounts mapped to Ethereum addresses.

If we analyze the workflow, we see that every entry used as a parameter when executing the scripts should contain only unique entries, e.g., `<account, balance>`. This invariant is important because the scripts do not know how to handle duplicates. The default behavior is that the last entry will replace all previous ones.

Currently, we do not have any enforcement in the files for this, especially in `zend_to_horizen.py`, where we use a mapped Ethereum address provided outside the dumping process, which could mistakenly contain duplicate entries.

Recommendation: Consider adding a duplicate check for every entry read from a file passed as a parameter to the following scripts: `get_all_forger_stakes.py`, `setup_eon2_json.py`, and `zend_to_horizen.py`.

Horizen: We added the check in the `zend_to_horizen` part in commit df5a47fd.

For the other two, the `JSON.load` removes duplicates automatically and considers only the last entry, if one is present. We acknowledge that this can be a potential issue, but considering that the dump and list of stakes functionality will not produce duplicate entries, we are fine with it.

We have added a duplicate check also on commit 221c1b8e.

Cantina Managed: Verified the fix on `zend_to_horizen` and acknowledged the rest.

3.2.6 Precision loss while converting satoshi to wei in `zend_to_horizen` scripts

Severity: Low Risk

Context: zend_to_horizen.py#L46.

Description: The `satoshi_2_wei()` function in the `zend_to_horizen.py` migration script has a floating-point precision loss when converting large satoshi values to wei. This can result in incorrect balance calculations during the migration process, potentially leading to loss of funds or inaccurate vault allocations.

```
def satoshi_2_wei(value_in_satoshi):
    return int(round(SATOSHI_TO_WEI_MULTIPLIER * value_in_satoshi))
```

Proof of Concept: The function performs multiplication using floating-point arithmetic when `value_in_satoshi` is passed as a float. Python's floating-point representation cannot accurately represent all large integers, leading to precision loss:

```
SATOSHI_TO_WEI_MULTIPLIER = 10 ** 10

def satoshi_2_wei(value_in_satoshi):
    return int(round(SATOSHI_TO_WEI_MULTIPLIER * value_in_satoshi))

**Expected: 210000000000000000000000000:**

# Actual:   209999999999999999110807552
print(satoshi_2_wei(21_000_000_0000_0000.0))
```

Recommendation: Consider fixing the issue by using `decimal` library as follows:

```
from decimal import Decimal

SATOSHI_TO_WEI_MULTIPLIER = Decimal('10000000000')

def satoshi_2_wei(value_in_satoshi):
    return int(Decimal(str(value_in_satoshi)) * SATOSHI_TO_WEI_MULTIPLIER)
```

Horizen: We have just removed the round in commit 060533edf considering satoshis can't be a decimal value and it's max value is `21_000_000 * 10^8`, is working fine without Decimal.

Cantina Managed: Verified fixes. Additionally, ensure the inputs are always provided without any floating-point values. E.g., providing inputs as `21_000_000e8` will lead to issues in Python.

3.3 Gas Optimization

3.3.1 Optimization in `ZendBackupVault` claims functions

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: The `ZendBackupVault` claims uses `_verifyPubKeysFromScript` which is verifying that the multisig script and the pubkeys are correct.

How it does it is by iterating through the script to extract each public key with these steps:

- Reads the size of the next public key.
- Validates that the key size matches either compressed (33 bytes) or uncompressed (65 bytes) format.
- Only attempts verification if the provided public key (x, y) coordinates are non-zero.
- Extracts the x-coordinate from the script using assembly.
- For uncompressed keys: extracts the full y-coordinate.
- For compressed keys: extracts the sign byte to determine y-coordinate parity.

For the compressed keys, the assembly code grabs the sign-byte in a round-about way:

- It loads a 32-byte word that begins 31 bytes before the real sign. (`script + 0x01 + pos`). After that, it keeps only the least-significant byte of the word.
- This byte coincides with `script[pos]` only because, in the current. script layout, the compressed-key prefix (`0x02 / 0x03`) happens to sit. at the end of that 32-byte word.

```
uint8 sign;
assembly {
    let resultPtr := mload(0x40)
    let sourcePtr := add(script, 0x01)
    let offset := add(sourcePtr, pos) //sign is at first byte

    mstore(resultPtr, mload(offset))
    sign := mload(resultPtr)
}
```

A more efficient way tho can get the sign more concise:

```
assembly {
         let offset := add(add(script, 0x20), pos) // data start + pos
         sign := byte(0, mload(offset))            // take the LS byte
}
```

Recommendation: Consider replacing that code with the optimized version.

Horizen: Fixed in commit 452b4361.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Minor code quality issues

Severity: Informational

Context: *(See each case below)*

Description: The following issues have been identified and aggregated, as they are minor and not worth reporting individually:

- `ZenToken.sol`:
  - ZenToken.sol?lines=7,7 — An interface should be used instead of the entire `LinearTokenVesting` contract to avoid cluttering the explorer.
  - ZenToken.sol?lines=4,4 -- Use `import "@openzeppelin/contracts/access/AccessControl.sol";` to maintain consistency with the other import statements.
  - ZenToken.sol?lines=22,22 -- Explicitly declare the visibility identifier for `numOfMinters` variable.
  - ZenToken.sol?lines=83,87 -- Consider replacing hardcoded values with constants.
  - ZenToken.sol?lines=20,24 -- Variables `horizenFoundationVested` and `horizenDaoVested` could be made immutable.
- `ZendBackupVault.sol`:
  - ZendBackupVault.sol?lines=4,4 — Use `import "./VerificationLibrary.sol";` to maintain consistency with the other import statements.
- `EONBackupVault.sol`:
  - EONBackupVault.sol#L4 — An interface should be used instead of the entire `ZenToken` contract to avoid cluttering the explorer.
- `ZenMigrationFactory.sol`:
  - ZenMigrationFactory.sol?lines=9,9 -- The `Strings` import is not used.

`ZendBackupVault.sol`: - ZendBackupVault.sol?lines=7,7 -- An interface should be used instead of the entire `LinearTokenVesting` contract to avoid cluttering the explorer. - ZendBackupVault.sol?lines=139,139 -- The EIP link should be replaced with `https://github.com/ethereum/ercs/blob/master/ERCS/erc-55.md`.

- `VerificationLibrary.sol`:
  - VerificationLibrary.sol?lines=28,28 -- The comment is wrong should be `1,32`.

Recommendation: Consider fixing these small issues.

Horizen: Fixed in commit 083abf19.

Cantina Managed: Fix verified.

3.4.2 Various missing events or events issues

Severity: Informational

Context: *(See each case below)*

Description: The following are issues related to events, either missing or needing improvement:

- There are several instances where events are missing despite state changes. Events should always be emitted to enable easy off-chain tracking. For example, in EONBackupVault.sol?lines=75,75, the `batchInsert` function does not emit any event. Please review all state-changing functions and ensure appropriate events are emitted.
- LinearTokenVesting.sol?lines=22,22 — `newBeneficiary` and `oldBeneficiary` should be indexed; typically, `address` topics should be indexed.
- LinearTokenVesting.sol?lines=21,21 — This should also include a `beneficiary` topic and the `msg.sender`.
- ZendBackupVault.sol?lines=60,60 -- The `Claimed` should include the `msg.sender`.

Recommendation: Consider addressing these issues to improve off-chain data tracking.

Horizen: Fixed in commit 0ac6feb9.

Cantina Managed: Fix verified.

3.4.3 The `AccessControl` in `ZenToken` is obsolete

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The use of `AccessControl` in `ZenToken` is unnecessary and may lead to confusion in the future. Its only intended purpose is to grant vaults permission to mint during airdrops. Once a vault completes its airdrop, it marks itself as finished and removes its minter role. There is no real benefit to using `AccessControl` in this case, especially since an ACL system is not required for `ZenToken` outside of minting. If retained, `AccessControl` will only introduce unused functions that add clutter to the contract.

Recommendation: Consider replacing `AccessControl` with a simple `mapping` of minters, which is updated to `false` once a minter completes its task:

```
// Simple mapping to track authorized minters
mapping(address => bool) public minters;

function notifyMintingDone() public canMint {
    // Remove caller from minters mapping
    minters[msg.sender] = false;
    unchecked {
        --numOfMinters;
    }
    ...
}
```

Horizen: Fixed in commit 29620109.

Cantina Managed: Fix verified.

3.4.4 The `distribute` of `ZEN` in the `EON` vault should not be capped to 500

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `distribute` function in `EONBackupVault` uses a hardcoded cap of 500 addresses per distribution.

```
function distribute() public onlyOwner {
    if (cumulativeHashCheckpoint == bytes32(0)) revert CumulativeHashCheckpointNotSet();
    if (address(zenToken) == address(0)) revert ERC20NotSet();
    if (_cumulativeHash != cumulativeHashCheckpoint) revert CumulativeHashNotValid(); // Loaded data not
    ↪ matching distribution locked
    if (nextRewardIndex == addressList.length) revert NothingToDistribute();

    uint256 count = 0;
    uint256 _nextRewardIndex = nextRewardIndex;
    while (_nextRewardIndex != addressList.length && count != 500) {
        address addr = addressList[_nextRewardIndex];
        uint256 amount = balances[addr];
        if (amount > 0) {
            balances[addr] = 0;
            zenToken.mint(addr, amount);
        }
        unchecked {
            ++_nextRewardIndex;
            ++count;
        }
    }
    nextRewardIndex = _nextRewardIndex;
    if (nextRewardIndex == addressList.length) {
        zenToken.notifyMintingDone();
    }
}
```

While this is likely sufficient and unlikely to run out of gas, it would be safer and more intuitive to allow the owner to specify a `maxCount` rather than using a hardcoded value.

Recommendation: Add a `maxCount` parameter to the `distribute` function to replace the hardcoded 500. This allows flexibility in case 500 iterations become problematic due to gas limits, and also enables batching more than 500 addresses when possible to reduce overall transaction costs.

Horizen: Fixed in commit 465c9c34.

Cantina Managed: Fix verified.

3.4.5 The `admin` and `owner` usage is confusing in `LinearTokenVesting`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `LinearTokenVesting` contract uses both `Ownable` and a separate `admin` field. The only function accessible to the owner is `setERC20`, which is called by the factory and cannot be called again. Other functions are gated by the `admin` field, such as:

```
function changeBeneficiary(address newBeneficiary) public isAdmin
// ...
function changeVestingParams(uint256 newTimeBetweenClaims, uint256 newNumberOfIntervalsToClaim) public isAdmin
```

Additionally, the `admin` field is declared as `immutable`, suggesting it was not intended to change:

```
address public immutable admin;
```

This dual-control model is confusing and unnecessary for the use case.

Recommendation: Remove the `admin` field and rely solely on `owner`. You can limit ownership transfers using a counter, allowing only two transfers (one from the OpenZeppelin constructor, and one from the factory):

```
function _transferOwnership(address newOwner) internal override {
    if (_ownershipTransferCount == 2) revert ImmutableOwner();
    address oldOwner = _owner;
    _owner = newOwner;
    _ownershipTransferCount++;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Horizen: Fixed in commit 60180703.

Cantina Managed: Fix verified.

3.4.6 Add `zenToken` check to `moreToDistribute()` function

Severity: Informational

Context: EONBackupVault.sol#L115

Description: The `moreToDistribute()` function in the `EONBackupVault.sol` contract requires an additional check for `address(zenToken) != address(0)` to ensure consistency with the `distribute()` function. While the `distribute()` function correctly checks if `address(zenToken) == address(0)` and reverts with **ERC20NotSet** if true, the `moreToDistribute()` function does not include this check. This creates an inconsistency where:

- `moreToDistribute()` can return true, indicating distribution is possible.
- But `distribute()` would revert with ERC20NotSet when called.

Recommendation: Add the `address(zenToken) != address(0)` check to the `moreToDistribute()` function:

```
function moreToDistribute() public view returns (bool) {
    return address(zenToken) != address(0) &&
           _cumulativeHash != bytes32(0) &&
           _cumulativeHash == cumulativeHashCheckpoint &&
           nextRewardIndex < addressList.length;
}
```

Horizen: Fixed in commit 362b9d83.

Cantina Managed: Fix verified.

3.4.7 Merkle Trees could be used instead of batch insert in the `ZendBackupVault`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `ZendBackupVault` uses the same strategy as the EON vault: we insert every address on-chain by calling `batchInsert`. However, it differs from the EON vault in that users must claim their tokens using signatures. This approach is somewhat inefficient and could be improved by using a Merkle Tree strategy, where no addresses are inserted on-chain. Instead, we create a Merkle Tree with all the addresses and their corresponding balances, then store the root—just as we currently store the `cumulativeHashCheckpoint`. We would then provide the user with the Merkle proof needed to claim their tokens. This would be a more efficient method that avoids the on-chain batch insertion.

Recommendation: Consider replacing the current approach with a Merkle Tree-based strategy.

Horizen: Acknowledged the potential for increased efficiency, as noted in the audit finding, and has been considered. Nevertheless, the existing implementation will be preserved to ensure operational similarity with EON.

Cantina Managed: Acknowledged.

3.4.8 S-malleability in the claiming process

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `ZendBackupVault` is using `claimP2PKH` / `claimP2SH` to claim the tokens by using a signed message from the ZEND mainchain addresses. The `claimP2PKH` / `claimP2SH` accept any 65-byte Zcash/Horizen ECDSA signature and pass it to `VerificationLibrary.parseZendSignature`, then to `ecrecover`.

Because the code never checks that the s value is low ($\leq n/2$) it will also accept the "high-s" twin ($s' = n - s$) that signs the very same message. An attacker who already owns one valid signature can therefore create another signature to pass a specific threshold check.

- **Why it is not exploitable in this contract:**
  - In the `claimP2PKH`: once the first accepted signature moves the full amount to `destAddress`, `balances[zenAddress]` becomes 0; every further claiming for this address will immediately reverts with `NothingToClaim`.
  - The attacker cannot redirect funds, because the message to sign commits to the chosen `destAddress`; flipping `s` does not alter that address.
  - The `claimP2SH` as well is safe because each entry in hexSignatures[] is tied to a fixed position in the redeem-script (_verifyPubKeysFromScript checks that pubKey[i] is exactly the key stored at offset i). You can only place the two variants in the slot that belongs to that key; the other slots expect different public keys and the signature twin will not verify against them.

Recommendation: For completeness and future-proofing normalize signatures by rejecting any with `s > SECP256K1_N/2`, e.g.:

```
if (uint256(signature.s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0)
    revert InvalidSignature();
```

Same as OpenZeppelin does.

Horizen: Fixed in commit 96c7a935.

Cantina Managed: Fix verified.