OpenZeppelin | security

Fireblocks Vesting Vault Audit

Fireblocks

July 10, 2025

Table of Contents

Summary

Type	Stablecoins
Timeline	From 2025-06-03 To 2025-06-10
Languages	Solidity

Total Issues	34 (24 resolved, 4 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	3 (3 resolved)
Low Severity Issues	12 (6 resolved, 3 partially resolved)
Notes & Additional Information	19 (15 resolved, 1 partially resolved)

Scope

OpenZeppelin audited the fireblocks/fireblocks-smart-contracts repository at commit 0035fe7, in particular pull request 13.

In scope were the following files:

```
contracts/
├── library
│   ├── Errors
│   │   └── LibErrors.sol
│   └── Utils
│       └── SalvageCapable.sol
└── vaults
    ├── interface
    │   ├── IVestingVault.sol
    │   └── IVestingVaultErrors.sol
    └── VestingVault.sol
```

Update: *During the fix review, changes were made to the `SalvageCapable` contract to bring it in line with its OFT implementation in pull requrest #35 at commit 22e4f9a. All resolutions and the final state of the audited codebase mentioned in this report are contained in commit a8f454f.*

System Overview

Pull request #13 adds the implementation of a token vesting vault, the `VestingVault` contract. It also includes the `IVestingVaultErrors` and `IVestingVault` interfaces, as well as a non-upgradeable base salvage capability contract, the `SalvageCapable` contract.

The system enables management of token vesting schedules for beneficiaries, supporting creation, claims, cancellations, and salvage operations under role-based access control. It aims to provide flexible token vesting, potentially as part of a broader token management or treasury framework.

At its core, the system supports vesting schedules composed of multiple periods, each with customizable start and end times, optional cliff periods, and specific token amounts. This enables intricate vesting structures tailored to diverse needs. Beneficiaries have granular control over their rewards, with the ability to claim vested tokens across all schedules, per individual schedule, or for specific periods within a schedule.

The contract operates in two distinct modes: global and individual, offering versatility in how vesting timelines are defined. In global vesting mode, all schedules are anchored to a shared start time set by the `startGlobalVesting` function, which is ideal for synchronized vesting across multiple beneficiaries. Individual mode uses absolute timestamps for each schedule. Governance is controlled through role-based access, with distinct roles for vesting admins, forfeiture admins, and salvage operators. Schedules are tracked with unique IDs stored in mappings, and each beneficiary's schedule IDs are maintained in an array, enabling efficient querying and management.

The forfeiture mechanism allows cancellable schedules to be terminated, distributing vested tokens to beneficiaries while returning unvested tokens to admins—balancing fairness with administrative control. The contract incorporates strict checks to avoid over-commitment of tokens. The `committedTokens` counter tracks the total tokens allocated to active schedules, and functions like `getAvailableBalance` provide transparency into the contract's capacity to support new schedules. Designed as non-upgradeable, the contract emphasizes immutability, reducing risks associated with post-deployment modifications.

Security Model and Trust Assumptions

The protocol uses the implementation of OpenZeppelin's `AccessControl` library for role-based access control. The different sensitive functionalities are divided into the following roles:

- `DEFAULT_ADMIN_ROLE`: The default administrator of the Vesting Vault contract. This role is assigned at deployment and can manage all other roles. It cannot be renounced.
- `VESTING_ADMIN_ROLE`: Assigned at deployment and later by the `DEFAULT_ADMIN_ROLE`. Can create vesting schedules, initiate global vesting, and release vested tokens to beneficiaries.
- `FORFEITURE_ADMIN_ROLE`: Optional; must be manually assigned by the `DEFAULT_ADMIN_ROLE`. Allows cancellation of vesting schedules. Notably, this role has the authority to retain most tokens from a vesting schedule upon cancellation and should be assigned with caution.
- `SALVAGE_ROLE`: Optional; must be manually assigned by the `DEFAULT_ADMIN_ROLE`. Allows salvaging of tokens (ERC-20, ERC-721) and leftover gas.

All roles are assumed to be held by trusted entities acting in the best interest of the system.

Integration

The following are important considerations for future integrations, divided into generic and ERC-20-specific integration concerns.

Generic Integration

- The current implementation cannot receive ETH, as it does not implement a `receive` or `fallback` function. Additionally, it cannot receive ERC-721 tokens directly, as it lacks the `onERC721Received` hook. However, native tokens (e.g., ETH) can still be received via `selfdestruct`, and ERC-721 tokens indirectly through `mint` or `transferFrom` methods. Integrators should be aware of these limitations.
- Sending ETH to `msg.sender` could trigger reentrancy via a fallback function. While this is currently safe in the `salvageGas` method due to access control and the lack of reentrancy impact, issues could arise if a child contract overrides behavior or makes ETH integral to vesting logic. To prevent unintended behavior—especially where ETH balances influence logic or state—inheriting contracts should disallow reentrancy.

- In complex systems where the `VestingVault` contract receives tokens during deployment, initial roles may not be granted correctly. Since `_grantRole` does not revert upon failure, this could result in tokens being locked in the contract with no administrator capable of recovering them. Integrators should ensure proper role assignment before sending any tokens.

- Since `VestingVault` uses OpenZeppelin's `Context._msgSender()`, integrators of meta transactions using paymasters or trusted forwarders for gasless transactions should ensure the forwarder accurately passes the intended sender's address to maintain secure beneficiary and admin operations.

- The contract uses internal functions like `_getClaimableAmountForPeriod` to calculate claimable rewards without directly modifying state, relying on current state variables (e.g., `period.claimedAmount`, `block.timestamp`) for accuracy. Integrators overriding these functions should ensure corresponding state updates are performed before invoking this function again, to prevent reporting stale values.

ERC-20 Integration

- Validation checks in the constructor aim to assert that the asset address passed corresponds to an ERC-20 token. However, checking for the `decimals` function alone does not guarantee that the address is a valid ERC-20 token. Many non-ERC-20 contracts expose `decimals` methods for unrelated purposes. Manual verification of the token address is recommended.

- If the inheriting contract intends to distribute tokens to multiple beneficiaries, and the underlying token includes mechanisms such as blocklists, a blocked beneficiary could prevent distribution to all recipients. This risk should be accounted for in the integration design.

- If the underlying token exists at multiple addresses, administrators might gain indirect access to transfer out vesting tokens.

- Rebasing tokens should be avoided, as they are not supported (this limitation is documented).

- As a general recommendation, avoid sending non-vesting tokens to the vault.

Medium Severity

M-01 `period.claimedAmount` Can Be Non-Zero on Schedule Creation

The `createSchedule` function in the `VestingVault` contract allows administrators to add vesting schedules for beneficiaries. The function validates `VestingPeriod` inputs to set up vesting schedules. However, it does not check that `claimedAmount` is zero, enabling a non-zero value to offset the beneficiary's vesting amount, thereby reducing or nullifying their receivable tokens.

Consider adding a check to ensure `claimedAmount` is zero for each period.

***Update:** Resolved in pull request #30 at commit da736e0. The `VESTING_ADMIN_ROLE` admin no longer passes `claimedAmount` when creating a schedule.*

M-02 Default Admins Can Still Renounce Their Role

The `VestingVault` contract uses OpenZeppelin's `AccessControl` to manage administrative roles, with `DEFAULT_ADMIN_ROLE` serving as the top-level role that controls all others. The `renounceRole` function has been overridden to prevent default admins from renouncing their role and potentially leaving the contract without administrative control. However, default admins can still revoke their own `DEFAULT_ADMIN_ROLE` using `revokeRole`, which is not currently overridden to disable this operation. This effectively bypasses the safeguard intended by overriding `renounceRole`.

To fully enforce this restriction, consider overriding `revokeRole` as well to prevent default admins from revoking their own role.

***Update:** Resolved in pull request #29 at commit 11c6150. It is worth noting that a non-authorized user calling the overridden function will also trigger the same error, even though they do not have the role.*

M-03 Global Vesting Mode Could Prevent Collecting the Vest

The `VestingVault` contract includes a global vesting mode feature. When enabled, the contract calculates reward distribution based on the `globalVestingStartTime` timestamp, which is set when an admin calls the `startGlobalVesting` function.

In global vesting mode, the `createSchedule` function does not validate the `startPeriod` timestamp. If an admin mistakenly uses absolute timestamps when creating a schedule before activating the global vesting, the effective vesting start time, calculated as `globalVestingStartTime + startPeriod`, may be set far in the future, unintentionally delaying or even preventing beneficiaries from claiming their rewards.

Consider validating the `startPeriod` value when global vesting mode is active, or at a minimum, provide thorough documentation to ensure the feature is used correctly.

Update: *Resolved in pull request #30 at commit a4daf4e. Two new caps have been placed: a `MAX_RELATIVE_TIME_THRESHOLD` cap on the period's `startPeriod` and a `MAX_DURATION` cap on the period's duration.*

Low Severity

L-01 Inconsistent Order of Operations

The purpose of the `SalvageCapable` contract is to provide a salvaging mechanism that enables the transfer of assets away from the contract itself. This is useful for recovering assets that would otherwise remain locked within the contract.

This functionality is implemented for ERC-20 tokens, the native currency, and ERC-721 tokens. However, there is an inconsistency in the order in which salvage events are emitted across these implementations. Specifically, in the ERC-721 implementation, the salvage event is emitted after the asset transfer is performed (which includes its own `Transfer` event), whereas in the other implementations, the salvage event is emitted before the transfer.

Such inconsistency can disrupt assumptions made by off-chain indexers regarding the expected order of events.

Since it is logically sound for the salvage event to follow the actual transfer of assets, consider aligning all implementations to follow a consistent event emission order. This ensures that off-chain components can reliably interpret and respond to the sequence of events.

Update: *Resolved in pull request #35 at commit 5f1f345. The order of the ERC-721 salvaging functionality has been modified to be consistent with the other implementations.*

L-02 Different Pragma Directives Are Used

In order to clearly identify the Solidity version with which the contracts will be compiled, pragma directives should be fixed and consistent across file imports.

Throughout the codebase, there are various pragma directives:

- `SalvageCapable.sol` has the pragma directive `pragma solidity ^0.8.22;`.
- `VestingVault.sol` has the pragma directive `pragma solidity 0.8.29;`.
- The `LibErrors.sol` library has the pragma directive `pragma solidity ^0.8.20;`.

Consider using the same fixed pragma version in all files.

Update*: Partially resolved in pull request #35 at commit 22e4f9a. The pragma directive of* `SalvageCapable.sol` *was modified to* `pragma solidity 0.8.29;` *as part of the unification with the implementation from the OFT Adapter. The Fireblocks team stated:*

> *Considering Solidity releases until today, the changes in* `LibErrors` *do not have any language language/compiler breaking changes that affect the functionality in these code files. LibErrors is only an error declaration file, and it is used by other contracts in this repo targetting lower patch versions, hence the need for keeping its pragma directives as-is.*

L-03 Misleading Documentation

Throughout the codebase in the `VestingVault` contract and `IVestingVault` interface, some comments inaccurately describe validation checks, token flow, and role definitions, which could potentially confuse developers.

For example:

- In the `VestingVault` contract: `"@param vestingToken_ The ERC20 token to be vested"` suggests `vestingToken_` is validated as an ERC-20 token, but the check `(address(vestingToken_).code.length == 0)` only verifies it's a contract.
- In the `IVestingVault` interface at the `VestingScheduleCancelled` event: `"@param reclaimedAmount The amount of tokens reclaimed by the contract"` implies tokens are retained by the contract, but they are transferred to the admin.
- In the `IVestingVault` interface at the `Schedule` struct: `"@param beneficiary The owner of the schedule"` describes the beneficiary as the schedule's owner, but it has no control since it's only the recipient of vested tokens.

Consider updating the misleading comments to accurately reflect the checks, token flow, and role definitions.

Update: *Resolved in pull request #36 at commit 54990cd.*

L-04 Require Statement With Multiple Conditions

Within the `VestingVault` contract, the `require(success && data.length > 0, LibErrors.InvalidImplementation())` statement has multiple conditions that must be satisfied.

To simplify the codebase and provide more helpful error messages for failing `require` statements, consider using a single require statement for each condition.

Update*: Acknowledged, not resolved. The Fireblocks team stated:*

> *In this particular situation (low-level call), we are comfortable with the multiple conditions since the revert reasons are highly correlated.*

L-05 Inconsistent Handling of Cancelled Schedules

The `VestingVault` contract allows claiming and releasing tokens from vesting schedules. The `claim(uint256)` and `release(uint256)` functions process cancelled schedules differently from the similar functions `claim` and `release`, relying on `VestingPeriod`'s `claimedAmount` equaling amount to return zero claimable tokens, which is misleading and results in unclear errors.

Consider adding checks for the `isCancelled` flag in these functions to revert with a clear `CancelledSchedule` error, ensuring consistent behavior and easier maintenance.

Update: *Resolved in pull request #33 at commit 778f34e.*

L-06 Vesting Token Assignment Cannot Be Reserved

To create a vesting schedule, a holder must first deposit `vestingToken` funds in the `VestingVault` contract. Then, the `VESTING_ADMIN_ROLE` can call the `createSchedule` function, passing the schedule's parameters, particularly the periods. Once the value of those periods is added, the contract validates whether the current available balance of the `vestingToken` funds, calculated as the difference between the current balance and the balance already committed by other schedules, is sufficient.

However, since this commitment of the tokens is not atomic, several situations could arise. In particular:

- As the `DEFAULT_ADMIN_ROLE` admin can add additional `VESTING_ADMIN_ROLE` admins, a lack of synchronization between them could result in a schedule B committing a batch of tokens that were meant for schedule A by mistake.
- If the `vestingToken` incurs a fee on transfer, `VESTING_ADMIN_ROLE` admins might account for the value sent by the holder instead of the value received in the vault. For example, if two holders each send 100 tokens to the vault to vest schedules A and B, and there is a 1-token fee on transfer, the `VESTING_ADMIN_ROLE` admin might create schedule A using 100 tokens, resulting in the commitment of 1 token belonging to future schedule B.
- In the case of a malicious `VESTING_ADMIN_ROLE` admin, they could frontrun the call to create a schedule and commit tokens meant for a different schedule.

To prevent the possibility of committing tokens to a different schedule, consider implementing an atomic operation to fund the `VestingVault` contract and commit those assets to a specific schedule.

Update: *Partially resolved in pull request #31 at commit 0d0156d. The Fireblocks team has implemented the `BoundedRoleMembership` contract that extends the logic from the*

`AccessControl` contract and limits the `VESTING_ADMIN_ROLE` admins to 1. Even though this mitigates the communication problems between different admins, it still does not allow the reservation of funds to a particular schedule. Moreover, it might reintroduce the original effect if the single `VESTING_ADMIN_ROLE` admin uses a multisig, where a different set of signers could create a schedule that would use the funds meant for another schedule.

L-07 Ambiguous Error Handling

The `VestingVault` constructor checks that the `vestingToken_` address has deployed code using `vestingToken_.code.length > 0`. It reverts with `InvalidImplementation` if the address lacks code, misleadingly implying a faulty contract implementation rather than indicating no contract exists at the address.

Consider introducing a specific error to describe this specific case.

***Update**: Resolved in pull request #36 at commit 3f75e60.*

L-08 Claim Operations Monotonically Increase in Gas Cost

The `VestingVault` contract allows beneficiaries to accumulate multiple vesting schedules over time through the `createSchedule` function, with each schedule containing multiple periods. When a beneficiary calls the general `claim` function to claim all available tokens, the contract iterates through all schedules associated with that beneficiary and then through all periods within each schedule. The current implementation in the `claim` function processes every schedule and period, including those that may have already been fully claimed, are not yet claimable, or are cancelled, though it does skip cancelled schedules.

Since these computations apply to all schedules and periods including those that yield zero claimable tokens, long-term users who have accumulated many vesting schedules over time will face increasingly expensive claim operations, potentially reaching gas limits that make the `claim` function unusable. This forces users to resort to the more granular `claim(scheduleId)` or `claim(scheduleId, periodIndex)` functions, which defeats the purpose of having a convenient *claim all* option and creates a poor user experience.

To address this issue, consider adding pagination support to the general `claim` function, allowing users to specify a range of schedules to process in a single transaction. Implementing a more efficient data structure that tracks only active claimable schedules, or adding a cleanup mechanism that removes fully claimed and expired schedules from the beneficiary's active schedule list, could also be beneficial.

Update: *Resolved in pull request #34 at commit e6eaa8f. A per-schedule bitmap has been added and a limitation of up to 256 periods per schedule has been imposed. However, the `claim` and `release` functions that receive the period as input deviate from the other analogous functions, in that they do not restrict the operation if the bit for that period is already off. Moreover, OpenZeppelin's `BitMaps` contract should be used to handle these operations.*

L-09 Committed Balance Is Reduced Before Usage

The `VestingVault` contract's `_processTokenRelease` function decreases `committedTokens` before performing the external token transfer. This creates a vulnerability for tokens with reentrancy possibilities where the contract state shows tokens as no longer committed while they have not actually been transferred. An attacker, with vesting admin role, could trigger reentrancy during transfer to call `createSchedule` with the temporarily inflated balance, potentially creating schedules exceeding actual token holdings. This could lead to contract insolvency where some beneficiaries cannot claim their vested tokens.

To address this issue, consider performing the token transfer first, then updating `committedTokens` only after a successful transfer. This eliminates the reentrancy window and maintains accurate accounting.

Update*: Resolved in pull request #33 at commit 6846cd7.*

L-10 Inconsistent Behavior on Non-Existent Schedule

In the `VestingVault` contract, whenever a `schedule.id` is zero, some getters will return `0`, while others will revert. Similarly, the check `globalVestingMode && !globalVestingStarted` sometimes returns zero and sometimes reverts the transaction.

It is common practice for getter functions not to revert, instead returning zero for certain checks, unlike setter functions. However, given the inconsistency in the codebase, it is advisable to carefully review the behavior and improve consistency where needed.

***Update**: Partially resolved in pull request #33 at commit 0d6e38f. More documentation has been added for the `getClaimableAmount` and `getClaimableAmount` getters. The Fireblocks team stated:*

> *We have made changes to remain consistent with the premise that "It is common practice for getter functions not to revert, instead returning zero for certain checks". Changes made include documentation improvements, since we believe that for the `getSchedule(uint256 scheduleId)` getter, which is supposed to return an entity (Schedule struct), it is better to revert. This is already documented.*

L-11 Vesting Periods Can Overlap

The `VestingVault` contract allows defining vesting schedules with multiple `VestingPeriod` structs, added via the `createSchedule` function. These periods, intended to model flexible vesting like milestones or cliff-plus-linear vesting, only require that each period passes the `startPeriod < endPeriod` check.

However, this restriction allows overlapping periods within the same schedule, and it is possible to have equal periods in the same schedule. As all these periods and schedules will be iterated and used to calculate the vested balance of the beneficiary, reducing the vesting curve to its minimal expression would result in improved gas efficiency when claiming, increased readability and understanding of the overall vesting schedule, and a reduction of potential vesting-design mistakes due to the complexity of simultaneous contributing periods and schedules.

A more straightforward approach would be to allow only a single vesting period at a time, ensuring that no periods overlap within the same schedule. For more complex vesting curves and patterns, such as those that include an initial cliff or an instantaneous vesting portion, an additional field could be added to the `VestingPeriod` struct to specify a delay before linear vesting begins. Enforcing chronological ordering of periods in the `createSchedule` function would simplify validation by ensuring that the start and end times of adjacent periods do not overlap.

In order to simplify vesting calculations, improve gas costs, and enhance understanding of the current periods being vested, consider applying the aforementioned solutions.

***Update**: Acknowledged, not resolved. The Fireblocks team stated:*

> *This does not suit our design.*

L-12 Inefficient Claim Mechanism

The `VestingVault` contract enables the creation of multiple vesting schedules. At the same time, the `claim` function is designed to process all of a beneficiary's schedules and their corresponding periods in a single transaction.

The primary issue is that the `claim` function's loops are unbounded. As the number of schedules or periods per schedule grows, the function's gas cost increases. This could lead to transactions reverting due to out-of-gas errors, preventing beneficiaries from claiming their vested assets. For this reason, other versions of the claim function are specified:

- One that takes a specific `scheduleId` and claims for all the periods within that specific schedule. This function does not entirely resolve the issue, since schedules with a high number of periods can be created. Given the different nature of checks within the `claim` function compared to the schedule creation, a transaction might still revert due to out-of-gas errors in the claim process.

- One that takes a specific `scheduleId` and a specific `periodIndex` to claim. This might create a scenario in which period indexes are not processed in order. While this is not a real issue, a specific order might be needed in the future.

Since the contract has no multicall feature, consider adding a functionality to specify an absolute number of periods to process within a claim and to have the contract follow an ordered claiming process through the list of periods for a specific schedule.

***Update**: Acknowledged, not resolved. The Fireblocks team stated:*

> *After prototyping a few solutions for tracking which schedules/periods where inactive (nothing else claimable), we were not able to find a solution that:*
>
> - *decreased the gas cost for the complex scenarios, without increasing it for the simpler (and expectedly the most common) scenarios*
> - *did not require enforcing chronological ordering of periods for either schedule creation or claiming*

Notes & Additional Information

N-01 The Consistency of `msg.sender` Usage Must Be Checked

The `VestingVault` contract inherits from the `Context` contract, which provides `_msgSender` and `_msgData` functions. In cases where the system allows meta transactions, these functions are usually overridden to extract the original message sender or data from it.

The usage of `msg.sender` in line 266 of the `VestingVault` is inconsistent with other parts of the codebase where `_msgSender` is used instead.

Consider making the usage of `msgSender` consistent so that, in cases where the contract enables meta transactions, consistent values are referenced for the message sender.

***Update**: Resolved in pull request #34 at commit 9211925.*

N-02 Typographic Error

The phrase **"RBAC control"** in `SalvageCapable` comments is redundant because "RBAC" (Role-Based Access Control) already includes "Control."

For example, incorrect comments are present above these functions:

- The _authorizeSalvageERC20 function
- The _authorizeSalvageGas function
- The _authorizeSalvageNFT function

Replace "RBAC control" with "RBAC" or "Role-Based Access Control" in the affected comments to eliminate redundancy.

***Update**: Resolved in pull request #35 at commit fb50e89.*

N-03 Invalid ERC-20 Check

The `VestingVault`'s `constructor` attempts to validate the `vestingToken` as an ERC-20 contract. However, it incorrectly assumes that tokens with zero decimals are invalid, which could lead to rejection of valid ERC-20 tokens that legitimately use zero decimals.

Consider revising the validation logic to properly detect ERC-20 compliance without excluding tokens based solely on their decimals value. This ensures compatibility with all ERC-20–compliant tokens.

***Update**: Partially resolved in pull request #36 at commit 690d30e. The validation is being performed on `totalSupply` instead of using the `decimals` method and value. However, this causes the same effect in true ERC-20 assets that might not have minted any token since deployment and are being added to the vesting before their first mint. Moreover, non-ERC20 contracts that implement the `totalSupply` method will likely also be able to pass the validation, even though they should not.*

N-04 Outdated Copyright Year in Contract License Headers

The `VestingVault` and `SalvageCapable` contracts, along with the `IVestingVault` and `IVestingVaultErrors` interfaces, include license headers containing copyright and licensing information. The headers specify a copyright year of 2024, which is outdated for contracts developed or released in 2025.

Update the copyright year to 2025 in both contracts' license headers to accurately reflect their current status.

***Update**: Resolved in pull request #34 at commit cfb401b.*

N-05 Implementation Inconsistency in `SalvageCapable`

The `SalvageCapable` contract enables the salvaging of ERC-20, ERC-721, and ETH assets sent to the contract. Unlike the ETH and ERC-721 salvage operations, the ERC-20 salvage operation uses a virtual internal `_withdrawERC20` function, creating an undocumented design inconsistency.

To ensure consistency, consider either documenting the rationale for making `_withdrawERC20` virtual or introducing similar `virtual internal` functions for ETH and ERC-721 salvaging operations.

***Update**: Resolved in pull request #35 at commit 4ae9a25.*

N-06 Data Type Mismatch in `getSchedule` Function Parameter

The `getSchedule` function accepts a parameter `scheduleId` of type `uint256`, while the corresponding `Schedule struct` defines its `id` field as a `uint32`. This inconsistency may lead to potential issues, such as implicit downcasting when a `scheduleId` falls outside the `uint32` range, causing unexpected behavior or errors in schedule management.

It is advisable to harmonize the data types between the function parameter and the struct field to ensure consistency and prevent logical errors.

***Update**: Resolved in pull request #34 at commit 5d536e5.*

N-07 Unused Errors

In the `LibErrors` library, the `AccountUnauthorized` and `RenounceRoleDisabled` errors are unused. The same is true for the `InvalidSchedule` error within the `IVestingVaultErrors` library.

To enhance the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused errors.

***Update**: Resolved in pull request #34 at commit 2d1d60c.*

N-08 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are contracts that do not have a security contact:

- The `SalvageCapable` abstract contract.
- The `IVestingVault` interface.
- The `IVestingVaultErrors` interface.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #34 at commit e0341fa and in pull request #35 at commit 22e4f9a.*

N-09 Unnecessary Unchecked Block

Starting from Solidity version 0.8.22, the compiler automatically optimizes arithmetic increments in for loops by removing overflow checks. The specific conditions in which that happens are better described in a separate article.

Within the `VestingVault` contract, the `unchecked{++i;}` pattern for index increments within `for` loops is present in several places.

To improve the overall clarity, intent, and readability of the codebase, consider removing the unchecked blocks considered unnecessary.

Update: *Resolved in pull request #34 at commit 5db0da2.*

N-10 Missing Named Parameters in Mapping

Since Solidity 0.8.18, mappings can include named parameters to provide more clarity about their purpose. Named parameters allow mappings to be declared in the form `mapping(KeyType KeyName? => ValueType ValueName?)`. This feature enhances code readability and maintainability.

In the `VestingVault` contract, the mapping state variables `scheduleById` and `beneficiaryToScheduleIds` do not have any named parameters.

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

Update: *Resolved in [pull request #34](#) at [commit d292bca](#).*

N-11 Indecisive Licenses

Throughout the codebase, there are multiple files with indecisive SPDX licenses:

`// SPDX-License-Identifier: AGPL-3.0-or-later`

Consider specifying only one license to prevent potential licensing ambiguity.

Update: *Acknowledged, not resolved. The Fireblocks team stated:*

> *We will not make any changes since this is our approved license identifier.*

N-12 Unclear Function Visibility

Throughout the codebase, all external functions are defined with the `override` keyword but not with the `virtual` keyword. This means they cannot be overridden by inheriting contracts, except for the `renounceRole` [function](#), which is explicitly marked as `virtual`.

Similarly, most internal functions are not defined as virtual, so they cannot be overridden in inheriting contracts (although they can still be called from them, since they are not `private`). The [exceptions](#) are `_authorizeSalvageGas`, `_authorizeSalvageNFT`, and `_authorizeSalvageERC20`, which are marked as `virtual` and can therefore be overridden.

To better reflect intentions and provide clearer guidance to users, consider documenting which functions are intended to be overridden. Additionally, review whether some internal functions should be declared private if they are not meant to be overridden or called by inheriting contracts.

Update: *Resolved in [pull request #34](#) at [commit baf2f73](#). Several functions have been marked as `virtual` while the visibility of some others has been reduced to `private`.*

N-13 Mixed Semantic Usage of Schedule IDs

The `id` parameter of the `Schedule` struct is assigned using a counter that is incremented before assignment. As a result, schedule IDs will start from `1`, effectively using the counter value as a unique identifier.

However, the identifier `0` is simultaneously used to signify the existence of any schedule, based on the assumption that the counter never generates a zero value.

To improve clarity and reduce semantic ambiguity, consider introducing a dedicated flag or `enum` value to indicate the existence of a schedule, rather than relying on the implicit meaning of a zero identifier. This approach would enhance the explicitness and maintainability of the code.

Update: *Acknowledged, not resolved. The Fireblocks team stated:*

> *We will not fix this as we believe that this semantic usage is reasonable when working with the EVM.*

N-14 Refactor Opportunities

Throughout the codebase, there are cases that could benefit from code refactoring. In particular:

- The check asserting that in "Global Mode" the global vesting has started is used in several functions and could be refactored as a modifier.
- In the `claim` function from the `VestingVault` contract, the call to the `_claim` function passes the `schedule` and the `scheduleId` as inputs. However, as the `schedule` already has the `id` as part of the struct, it does not need to be sent as another parameter.

Consider applying the aforementioned suggestions.

Update: *Resolved in pull request #34 at commits 6440c8d and 97da619. The Fireblocks team has implemented the `_validateGlobalVestingStatus` `internal` function instead of a modifier to ensure a consistent global mode status. Moreover, they have removed the `scheduleId` parameter from the `_claim` `internal` function.*

N-15 Redundant Public Visibility of `scheduleById`

The public visibility of the `scheduleById` mapping is redundant with the `getSchedule` function, which is designed to return the complete struct.

Consider changing the visibility of `scheduleById` to `internal` to remove the redundant getter and enforce the use of `getSchedule` for schedule retrieval.

***Update**: Resolved in pull request #34 at commit e0c2b74. Variable naming has been altered to reflect the `internal` visibility.*

N-16 Custom Errors in `require` Statements

Since Solidity version `0.8.26`, custom error support has been added to `require` statements. Initially, this feature was only available through the IR pipeline, but Solidity `0.8.27` extended support to the legacy pipeline as well.

Throughout the codebase, there is inconsistent use of `if-revert` and `require` statements. Additionally, many `if-revert` statements could be replaced with `require` statements for clarity and conciseness.

To improve readability, maintainability, conciseness, and gas savings, consider adopting only one approach and replacing `if-revert` statements with `require` ones.

***Update**: Resolved in pull request #34 at commit 247626a. The Fireblocks team stated:*

> *Some multi-condition checks maintained at most one condition in a wrapping `if` statement as we believe that this is clearer than a standalone multi-condition `require` statement.*

N-17 Strict Checks on Claimed Value Might Continue Vest

The `VestingVault` contract includes a check on line 996 to skip periods that have been fully claimed by comparing `period.claimedAmount == period.amount`. This check is designed to avoid unnecessary calculations for periods where all tokens have already been claimed.

However, the current implementation uses strict equality (==), which could potentially be bypassed in edge cases involving rounding errors. While the current implementation does not seem to have corner cases in which rounding errors can occur, using strict equality creates a potential vulnerability if future modifications introduce rounding operations or if there are any unforeseen edge cases in the arithmetic calculations. If the equality check fails, the function would continue processing and could potentially allow the claiming of tokens that should not be claimable, or at a minimum perform unnecessary gas-expensive calculations.

Following defensive programming practices, consider replacing the strict equality check with a greater-than-or-equal-to comparison to ensure the function properly handles any edge cases where `period.claimedAmount` might exceed `period.amount`.

Update*: Resolved in pull request #34 at commit 74b61aa.*

N-18 Claim Can Only Be Called by Beneficiary

The `claim` and `release` functions in the `VestingVault` contract distribute tokens to beneficiaries according to their configured schedules. However, these functions are restricted to either be called by the schedule's beneficiary or the `VESTING_ADMIN_ROLE` admin, reverting if any other address attempts to call the `claim` (or `release`) function on their behalf. Depending on their jurisdiction, beneficiary users might want to allow, or not, the possibility to use a different address for paying the claiming call.

In order to accommodate this flexibility, consider adding a flag during schedule creation that would allow or disallow third-party addresses to call the respective claiming or releasing functions on their behalf.

Update*: Acknowledged, not resolved. The Fireblocks team stated:*

> *This is by design.*

N-19 Inconsistent Usage of Return Variable

The `_getVestedAmountForPeriod` function in `VestingVault` calculates the vested token amount for a vesting period. This function inconsistently uses explicit `return` statements for different cases and the named return variable `vestedAmount` for linear vesting calculations.

Consider adopting explicit `return` statements for all conditions to ensure a consistent `return` method throughout the function.

Update: *Resolved in pull request #34 at commit e97b5a8. The named return variable has been removed from the aforementioned function. However, this causes an inconsistency with other functions.*

Conclusion

The audited codebase implements a system for managing token vesting schedules for beneficiaries, supporting the creation, claiming, cancellation, and salvage of schedules within a role-based access control model. It is designed to offer flexible token vesting functionality, potentially as part of a broader token management or treasury framework.

During the audit, several medium-severity issues were identified, including missing validation for zero claim amounts, improper timestamp handling in global vesting mode, and the ability for default admins to bypass role renouncement restrictions.

Additionally, several low-severity issues and informational notes were reported to improve code readability and facilitate future audits, integrations, and development.

The Fireblocks team was highly responsive and provided valuable support throughout the audit process, including access to documentation and specification materials that were essential for understanding the system.