

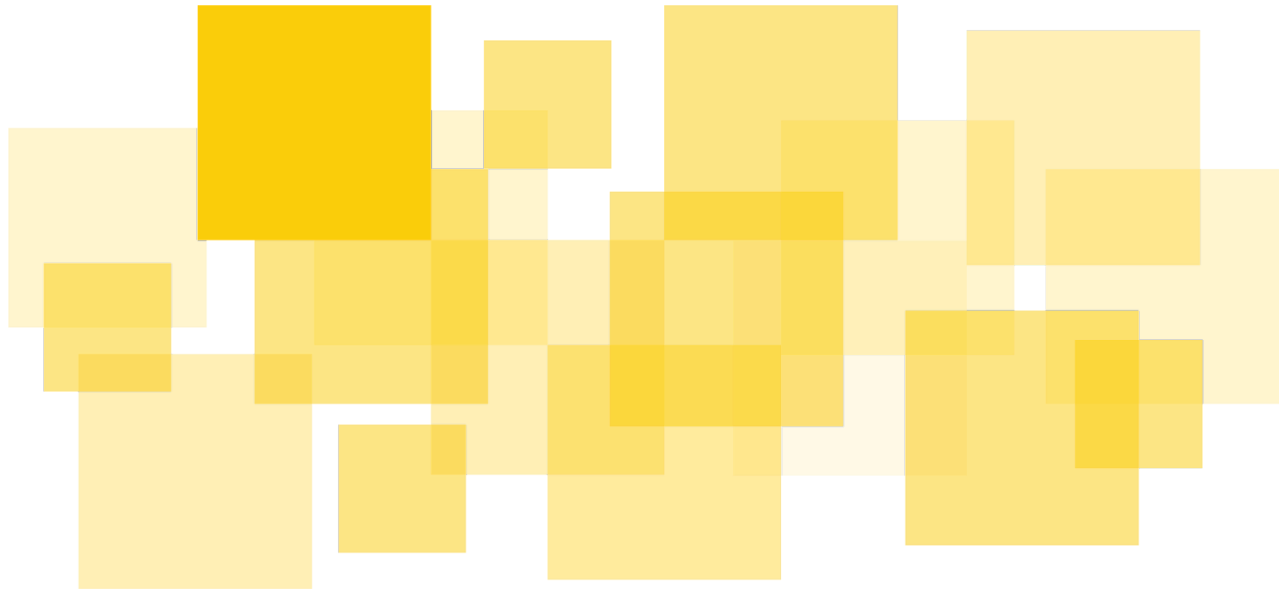


# Security Audit Report

---

TokenOps Stellar

Delivered: March 31, 2025



Prepared for Vesting Labs by





# Table of Contents

---

- [Disclaimer](#)
- [Executive Summary](#)
- [Goal](#)
- [Scope](#)
- [Methodology](#)
- [Platform Features and Logic Description](#)
  - [Token Vesting Factory](#)
  - [Token Vesting Manager](#)
  - [Sample workflow](#)
- [Invariants](#)
  - [Invariants provided by the client](#)
  - [Further invariants identified in the audit engagement](#)
- [Findings](#)
  - [\[A1\] Integer Overflow Possibility When Calculating Claimable Amounts](#)
  - [\[A2\] Function for Setting Administrators Can Suffer Denial of Service](#)
  - [\[A3\] Failure to properly execute the administrative access validations](#)
  - [\[A4\] Initialization of the Token Vesting Manager uses forwarded, non-verified parameters](#)
  - [\[A5\] Vesting Tokens Can Be Unlawfully Withdrawn By Administrators](#)
- [Informative Findings](#)
  - [\[B1\] Best practices recommendations](#)
  - [\[B2\] Misusage of Storage](#)
  - [\[B3\] Vesting operations can start in the past](#)
  - [\[B4\] Token contract calls are performed using env.invoke\\_contract](#)
  - [\[B5\] The Creation of Vesting Operations in Batch May Fail for Gas-Related Reasons](#)



## Disclaimer

---

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

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



# Executive Summary

---

Vesting Labs engaged Runtime Verification Inc. to conduct a security audit of the Token Vesting Factory and Token Vesting Manager contracts' code. The objective was to review the platform's business logic and implementation in Rust (Soroban) and identify any issues that could cause the system to malfunction or be exploited.

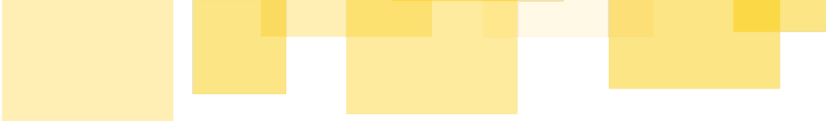
The audit was conducted over four calendar weeks (February 4, 2024, through March 4, 2025) and focused on analyzing the security of the source code of Vesting Labs' contracts for creating and managing token vesting operations. At a high level, these contracts enable users to create vesting operations on-chain, store their tokens, and claim them according to the business logic specified within them. Several other features are included in these contracts, which will be elaborated on later in this report.

The audit led to identifying issues of potential severity for the protocol's health, which have been identified as follows:

- Potential threats to users' fund integrity: [A1] Integer Overflow Possibility When Calculating Claimable Amounts, [A3] Failure to properly execute the administrative access validations, [A5] Vesting Tokens Can Be Unlawfully Withdrawn By Administrators;
- Potential code/logic malfunction: [A2] Function for Setting Administrators Can Suffer Denial of Service, [A4] Initialization of the Token Vesting Manager uses forwarded, non-verified parameters, [A5] Vesting Tokens Can Be Unlawfully Withdrawn By Administrators, [B2] Misusage of Storage, [B3] Vesting operations can start in the past, [B5] The Creation of Vesting Operations in Batch May Fail for Gas-Related Reasons;
- Administrative access bypass: [A3] Failure to properly execute the administrative access validations;

In addition, several informative findings and general recommendations have also been made, including:

- Best practices and code optimization-related particularities: [B1] Best practices recommendations, [B2] Misusage of Storage, [B4] Token contract calls are performed using `env.invoke_contract`, [B5] The Creation of Vesting Operations in Batch May Fail for Gas-Related Reasons;

- 
- Blockchain-related particularities: [\[B1\] Best practices recommendations](#), [\[B4\] Token contract calls are performed using env.invoke\\_contract](#), [\[B5\] The Creation of Vesting Operations in Batch May Fail for Gas-Related Reasons](#).

The client has addressed the majority of the findings listed in this report. The non-addressed findings have been deemed either harmless to the overall protocol's health and user safety or intended by design.



## Goal

---

The goal of the audit is threefold:

- Review the high-level business logic (protocol design) of Vesting Labs' system based on the provided documentation and code;
- Review the low-level implementation of the system for the individual Soroban smart contract (Token Vesting Factory and Manager);
- Analyze the integration between abstractions of the modules interacting with the contract in the scope of the engagement and reason about possible exploitative corner cases.

The audit focuses on identifying issues in the system's logic and implementation that could potentially render the system vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlights informative findings that could be used to improve the safety and efficiency of the implementation.



## Scope

---

The scope of the audit is limited to the code contained in a public [Github repository provided by the client](#). Within the repository and among other files, a contract is highlighted as in the scope of the engagement. The repository and contract are described below:

1. Token Vesting Factory:

- Deployer contract of Token Vesting Managers;
- `contracts/token_vesting_factory/src/lib.rs` , commit id `b820451ec996e9398d3b4207f5c9b30621c0f1ae` .

3. Token Vesting Manager:

- Contract that manages token vesting operations;
- `contracts/token_vesting_manager/src/lib.rs` , commit id `b820451ec996e9398d3b4207f5c9b30621c0f1ae` .

The comments provided in the code, a general description of the project, including samples of tests used for interacting with the platform, and online documentation provided by the client were used as reference material.

The audit is limited in scope to the artifacts listed above. Off-chain, auto-generated, or client-side portions of the codebase, as well as deployment and upgrade scripts, are not in the scope of this engagement.

Commits addressing the findings presented in this report have also been analyzed to ensure the resolution of potential issues in the protocol.



## Methodology

---

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in our [Disclaimer](#), we have followed the approaches described below to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the code, validating security-critical properties to ensure the absence of loopholes in the business logic. To this end, we carefully analyzed all the proposed features of the platform and the actors involved in the lifetime of deployed instances of the audited contracts.

Second, we thoroughly reviewed the contracts' source code to detect any unexpected (and possibly exploitable) behaviors. To facilitate our understanding of the platform's behavior, higher-level representations of the Rust codebase were created, where the most comprehensive were:

- Modeled sequences of logical operations, considering the limitations enforced by the identified invariants, checking if all desired properties hold for any possible input value;
- Manually built high-level function call maps, aiding the comprehension of the code and organization of the protocol's verification process;
- Performed fuzz testing with [Komet](#) to check if the specified invariants would hold for a set of randomized interactions with the contracts;
- Made use of static analyzers such as [Scout](#) to identify commonly identifiable issues;
- Created abstractions of the elements outside of the scope of this audit to build a complete picture of the protocol's business logic in action.

This approach enabled us to systematically check consistency between the logic and the provided Soroban Rust implementation of the system.

Once we completed higher-level abstractions and contract analysis, we systematically tested the protocol's business logic to identify potential invariant violations, leading to some of the findings in this report.

We also highlight that the findings discovered using the Scout tool, although valuable, have no context of the application or the business logic of the protocol that it analyzes precisely because of the tool's nature as a static analyzer. Therefore, findings have only been integrated into this report when applicable, and their severity level has been modified accordingly.





Finally, we conducted rounds of internal discussions with security experts over the code and platform design, aiming to verify possible exploitation vectors and identify improvements for the analyzed contracts. Code optimizations have also been discovered as an outcome of these research sessions and discussions.

Additionally, given the nascent Stellar-Soroban development and auditing community, we reviewed [this list](#) of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to the smart contracts and scripts; if they apply, we checked whether the code is vulnerable to them.



## Platform Features and Logic Description

---

Vesting Labs, the group responsible for the [TokenOps](#) project, focuses on simplifying and streamlining the complex process of managing token allocations and distributions, particularly for Web3 companies and projects. Token allocation occurs through vesting operations, where individuals or groups earn the right to receive allocated tokens or equity over time rather than all at once. This operation works as a delayed reward system, where instead of being able to claim all promised tokens upfront, the equity, or, in the case of TokenOps, tokens, are released in installments over time, often tied to certain milestones or time-based schedules.

Considering their vesting operations, Vesting Labs' platform allows tokens and details of the operations themselves to be tracked, ensuring transparency and accountability. The platform automates and manages token vesting schedules, releasing tokens over time according to pre-defined user agreements and incentivizing the long-term participation of the parties involved in the vesting operation. TokenOps also facilitates the distribution of tokens to various stakeholders, aiming to integrate with multiple blockchain networks and token standards. Furthermore, the platform offers reporting and analytics tools, giving companies valuable insights into token distribution and ownership structure. By streamlining these processes, TokenOps helps Web3 projects focus on development and community growth rather than getting lost in token management's intricacies while potentially aiding compliance efforts by maintaining a clear audit trail.

As mentioned in the Scope section of this report, two smart contracts were reviewed for the duration of this engagement; the Token Vesting Factory, and the Token Vesting Manager.

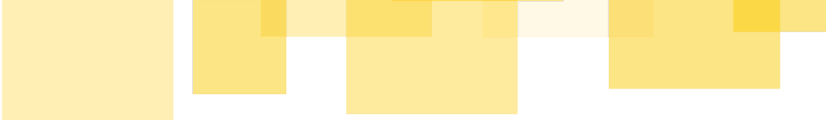
### Token Vesting Factory

---

The Token Vesting Factory is the deployer contract of the Token Vesting Manager instances. Whenever a user desires to create a contract with a specific set of administrators, which will also perform vesting operations using a specific token, the Token Vesting Factory is used to deploy a new manager contract.

All of the functions implemented in the Token Vesting Factory are public and thus available to be called by users. The functions are described below:

- `init` : receives an address that represents the owner of the contract and the hash of the wasm contract, which will be used to deploy the manager contracts. It is supposed to panic



if the storage entry for the owner address already exists, guaranteeing that this contract can only be initialized once. If not initialized, storage entries for the owner, for the manager contract wasm hash, and a 32-byte-vector counter auxiliary to the deployment of managers, referred to as salt, are created.

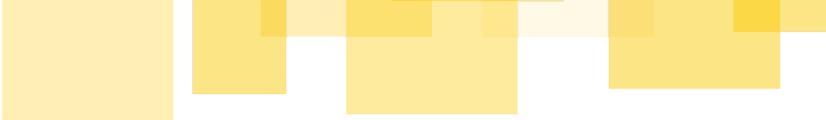
- `new_token_vesting_manager` : receives a vector of values which must contain the initialization args of the Token Factory Manager. It uses the stored wasm hash and the salt to deploy a new instance of the manager contract, forwarding the arguments received as a parameter to this function as the initialization function for the manager, appropriately updating the salt in preparation for the next deployment. This function can be called by any user, as there are no validations on the caller's address.
- `update_owner` : receives the address of the caller as a parameter and the address of the new owner. The caller must be at the same address as the stored owner's address. The new owner cannot be the same address as the current owner. If these conditions are respected, the storage entry for the owner is updated.
- `update_vesting_manager_wasm_hash` : receives the address of the caller and the new wasm hash for the manager contract. The caller must be the same address as the stored owner's address. The new wasm hash cannot be the same address as the current wasm hash. If these conditions are respected, the storage entry for the wasm hash is updated.
- `get_owner` : retrieves the storage entry for the owner of the Token Vesting Factory.
- `get_vesting_manager_wasm_hash` : retrieves the storage entry for the wasm hash for newly deployed manager contracts.

The Token Vesting Factory has no `upgrade` function implementation.

## Token Vesting Manager

---

The Token Vesting Manager is responsible for managing the vesting operations set up by a user or groups of users. Once initiated, it can hold the tokens supplied by an administrator of the contract and allow tokens to be claimed according to the business logic specified in the contract. Besides being able to configure the recipient of a vesting operation, the number of tokens to be vested, the time intervals in which the tokens are liberated for recipients, and the total duration of the vesting operation, the administrator of the manager can create vesting operations with a configurable initially unlocked amount, and a cliff amount. The initially unlocked amount is the number of tokens a recipient can claim after the start of the vesting operation. The cliff amount, similarly to the initially locked amount, is a number of tokens that



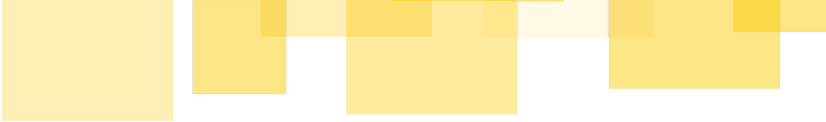
are made available to recipients of vesting operations once a certain amount of time has passed. While these two serve a similar purpose, they differ in functionality from a legal and contractual perspective.

The Token Vesting Factory immediately deploys and initiates an instance of the Token Vesting Manager, providing two addresses as the parameter for the manager's `init` function. The first one represents the address of the first administrator of this manager contract, while the second represents the address of a token following the SEP-41 token interface.

The administrator is capable of registering more administrators to the protocol, as well as modifying their administrative status with the `set_admin` function. The administrator addresses are stored in the `ADMINS` storage entry, which is reserved to a map that keeps the address to a boolean, representing its administrative status mapping. Once added to the map of admins of the manager contract, the address cannot be removed anymore. Still, its administrative status can be revoked by changing the mapping of the address to `false`. A counter on the number of active administrators is kept in the `ADMIN_COUNT` storage entry.

Administrators can create vesting operations through the manager contract by calling `create_vesting` function. It receives the following parameters:

- `caller` : address used to validate that caller is an administrator. `caller` is also validated to ensure that it has the same address as the one calling this function;
- `recipient` : address of the recipient of the vesting operation;
- `start_timestamp` : UTC timestamp of the start of the vesting operation;
- `end_timestamp` : UTC timestamp of the end of the vesting operation;
- `timelock` : UTC timestamp of the moment in time in which the vesting operation is not considered locked anymore, and attempts to claim tokens can be made;
- `initial_unlock` : number of tokens initially unlocked after the timelock period;
- `cliff_release_timestamp` : UTC timestamp of the moment in time in which the tokens reserved for the cliff can be made;
- `cliff_amount` : number of tokens reserved for the cliff, which can be claimed after the cliff release timestamp;
- `release_interval_secs` : interval in seconds where fragments of the total amount of vested tokens are released;
- `linear_vest_amount` : number of tokens that will be periodically released in the vesting operation, excluding the initially locked and the cliff amounts;

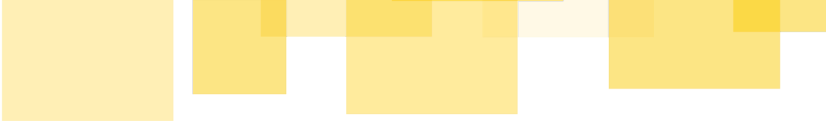


A few constraints must be respected in order to create a vesting operation successfully. Based on the above parameters, the constraints are:

- $linear\_vest\_amount + cliff\_amount \neq 0$  - Tokens must be reserved for this vesting operation;
- $start\_timestamp \neq 0 \wedge start\_timestamp \leq end\_timestamp$  - the start of the vesting operation can not be UTC timestamp 0, and the start of the vesting operation should be before the end timestamp;
- $release\_interval\_secs \neq 0$  - the interval in which tokens are released is used to calculate how many tokens can be claimed after a period of time. The interval cannot be zero so as not to cause division by zero issues;
- $if\ cliff\_release\_timestamp = 0\ then\ cliff\_amount = 0 \wedge (end\_timestamp - start\_timestamp) \bmod release\_interval\_secs = 0$  - ensures that no misconfiguration related to the non-existence of a cliff can happen. Also, the interval between the start and end of the vesting operation must be divisible with no precision loss to avoid possible miscalculations and rounding errors in the claimed amounts;
- $if\ cliff\_release\_timestamp \neq 0\ then\ cliff\_amount = 0 \wedge start\_timestamp \leq cliff\_release\_timestamp \leq end\_timestamp \wedge (end\_timestamp - cliff\_release\_timestamp) \bmod release\_interval\_secs = 0$  - ensures that if there is a cliff timestamp, then there must be a cliff amount to be released when that timestamp comes. Also enforces that the cliff happens within the duration of this vesting operation and that, since the cliff timestamp is used in place of the start timestamp when a cliff is used, its interval to the end timestamp must be divisible by the release interval with no precision loss to avoid possible miscalculations and rounding errors in the claimed amounts.

If these constraints are respected, an entry in the vestings map will be created, and a unique ID will be assigned to this vesting, storing information about this vesting operation. The number of tokens reserved for vesting, which works as an auxiliary balance to this contract, and any other entries potentially related to the vesting operations in this manager are updated. Finally, the amount of tokens to be vested is transferred from the caller to the Token Vesting Manager.

Several vesting operations can be created in a single transaction by calling the `create_vesting_batch` endpoint on the Token Vesting Manager. The parameters for the



individual vesting operations are passed to this endpoint through a structure ( `create_vesting_batch_params` ). This endpoint iterates over each entry in the lists of this struct, performing calls to `create_vesting_internal` and instantiating new vesting operations at each call.

Once created, a vesting operation can be revoked by an administrator of the Token Vesting Manager contract. The purpose of that is twofold. Firstly, if a misconfigured token vesting operation is created, administrators are able to "cancel" it and create it once again. Secondly, if one of the parties reaches a condition in which the vesting operation should not proceed anymore, it can be revoked.

Operations that are revoked can have their tokens claimed by their recipients at any point in time, but the tokens that vested until the point in which the operation was revoked can be claimed. The remaining tokens transferred to the manager can be claimed at any point in time.

The amount of vested tokens that cannot be claimed anymore once a vesting operation has been revoked is deducted from the `TOKENS_RESERVED_FOR_VESTING` storage entry. If the Token Vesting Manager holds more of the vesting tokens than the amount stored in `TOKENS_RESERVED_FOR_VESTING` , the administrators are allowed to withdraw them by calling the `withdraw_admin` endpoint. This way, the admins can claim back non-vested tokens from revoked operations and withdraw numbers of the vesting tokens arbitrarily sent to the manager (not sent through the `claim` endpoint).

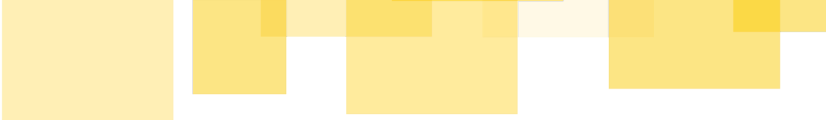
If, by chance, any token other than the vesting token is arbitrarily sent to the manager contract, an administrator can withdraw it using the `withdraw_other_token` endpoint.

Recipients can claim vested tokens by calling the `claim` endpoint of the manager contract. In order to receive their tokens, the vesting operation must not be within the timelock period. The amount that can be claimed per `claim` call is defined by the following equation:

$$Claimable = CL + IU + \frac{LVA \cdot \left\lfloor \frac{ST - ART}{RIS} \right\rfloor \cdot RIS}{ET - ST} - CA$$

Where the variables used in this equation are described below:

- **CL** (Cliff): token amount reserved for the cliff of the vesting operation;
- **IU** (Initial unlock): token amount provided in the vesting operation once the timelock period expires;

- 
- **LVA** (Linear vesting amount): the amount of tokens to be distributed within the duration of the vesting operation;
  - **ST** (Start timestamp): UTC timestamp of the vesting operation start;
  - **ET** (End timestamp): UTC timestamp of the vesting operation end;
  - **RIS** (Release Interval (Seconds)): interval in which fraction of the linear vested tokens are made available for claiming;
  - **CA** (Claimed amount): amount that has been claimed from the vesting operation so far.

Every time some tokens are claimed from the vesting operation, this specific amount is added to  $CA$ , which is then used in subsequent claim operations to keep track of how much has been withdrawn from the vesting and how much can still be claimed.

Besides the core functions described above, some peripheral functionalities are offered in the manager contract, including, but not limited to, functions for fetching information about administrators, recipients, vesting operations, vesting tokens, and the number of tokens reserved for vesting.

## Sample workflow

---

See the diagram in Figure 1 for the initialization of a Token Vesting Manager and the creation of a vesting operation.

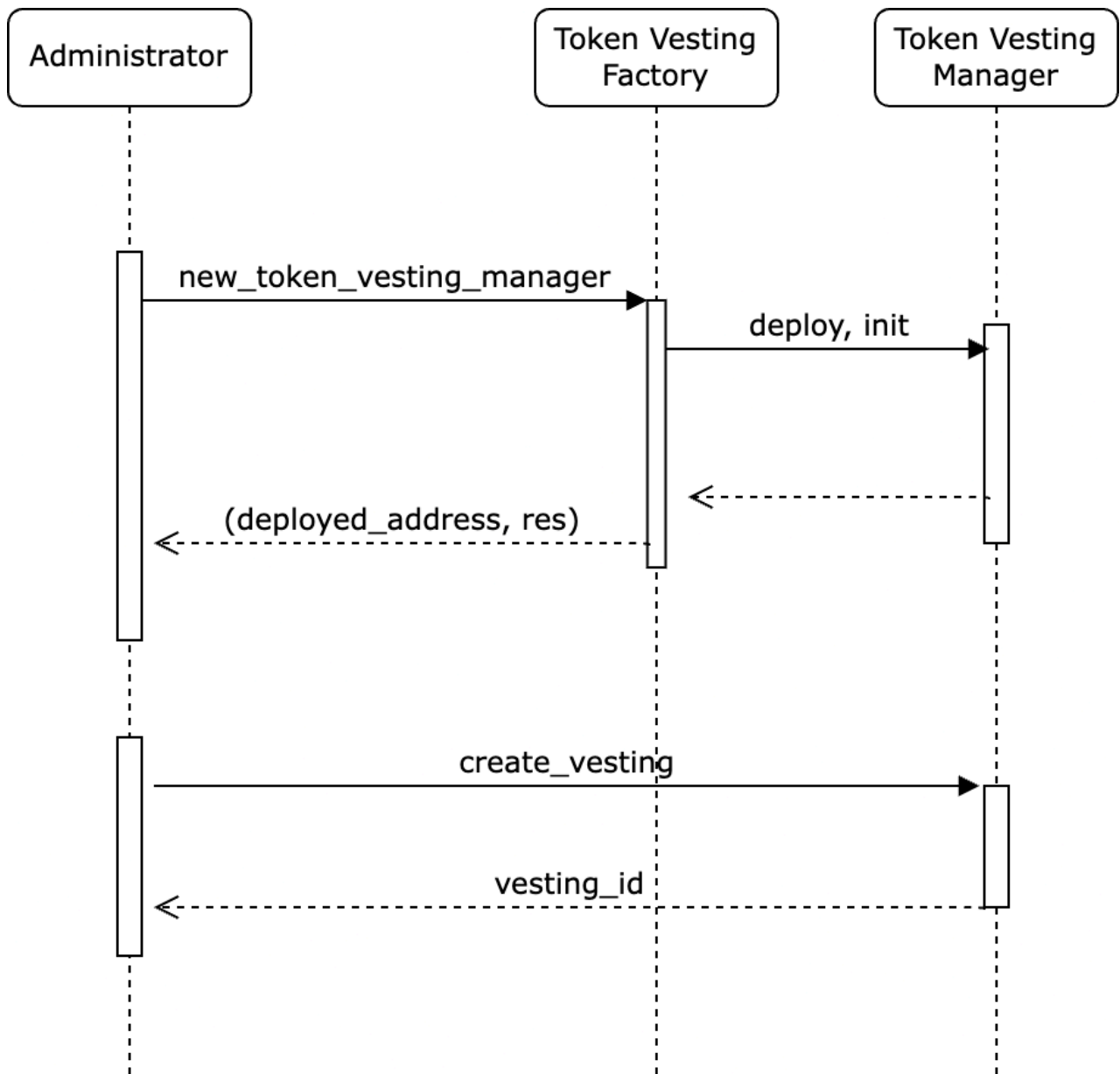


Figure 1: Deployment and initialization of the Token Vesting Manager. Initiating a token vesting operation.

The deployment and initialization of the Token Vesting Manager is atomic, and it is performed by the Token Vesting Factory. The caller becomes the administrator of the Token Vesting Manager and is allowed to add more administrators and create vesting operations. The diagram in Figure 1 displays the logic of using an already deployed Token Vesting Factory to instantiate a new



manager contract. Once created, the admin, as well as any other added administrator, is able to create as many vesting operations as required.

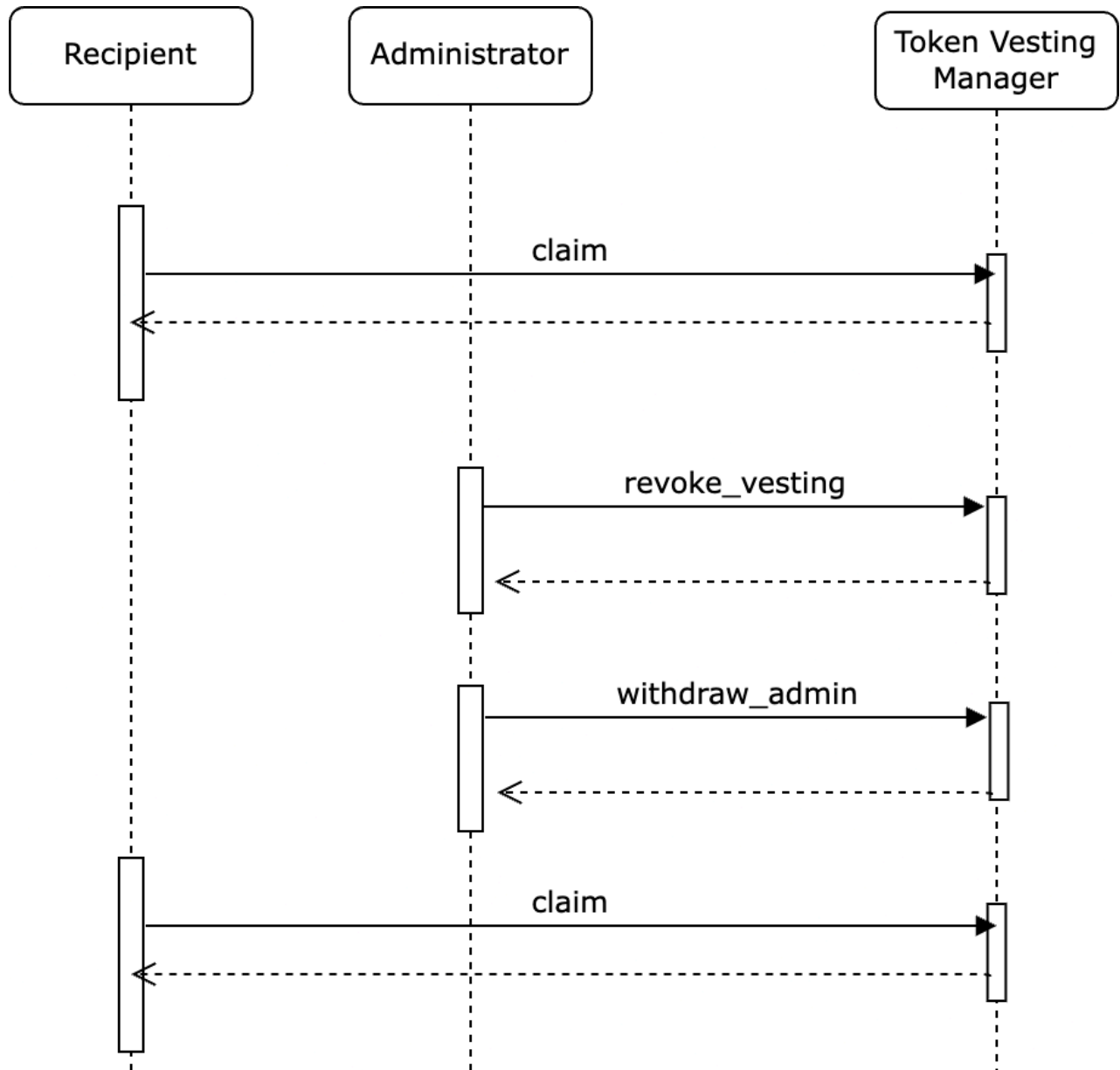


Figure 2: Usage of a Token Vesting Manager.

In Figure 2, we can see the logic of the utilization of a vesting operation handled by a Token Vesting Manager. A recipient can attempt to claim their tokens at any point in the lifetime of the



contract, only being able to if the conditions for vesting a number of tokens have been met. Administrators can revoke the vesting operations at any point, including after the end of the vesting operations. If tokens have not been vested at the point of the vesting's revokement, the administrator can withdraw them. If tokens have been vested prior to the vesting's revokement, the recipient can still claim them even after the revoking operation.



# Invariants

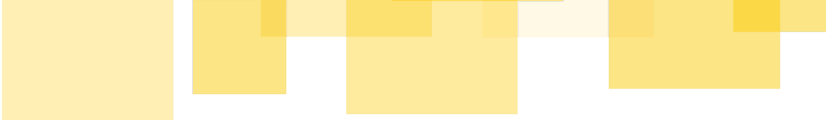
---

During the audit, invariants have been defined and used to guide part of our search for possible issues with TokenOps' contracts. Using the client's documentation, intended business logic, and references collected during the audit, we identified the following invariants:

## Invariants provided by the client

---

- Vesting token address cannot be changed;
- Vesting operations should have amounts to be distributed to recipients either in the linear vesting amount or the cliff amount ( `_linearVestAmount + _cliffAmount > 0` );
- Vesting operations should not start at the beginning of the UTC timeframe ( `_startTimestamp > 0 & _endTimestamp > 0` );
- Vesting operations should start before they end ( `_startTimestamp < _endTimestamp` );
- The interval in which tokens are vested has to be greater than 0 seconds ( `_releaseIntervalSecs > 0` );
- The internally kept balance of reserved tokens for vesting ( `TOKENS_RESERVED_FOR_VESTING` ) should be equal to the sum of all unclaimed tokens across all vesting schedules;
- The vesting token balance of the manager contract should always be greater or equal to the internally kept balance of reserved tokens for vesting ( `TOKENS_RESERVED_FOR_VESTING` );
- Each vesting operation must have a unique ID that is not reused for any other vesting;
- The claimed amount on a vesting operation should always be smaller or equal to the total amount vested in that operation, consisting of the linear vest amount, cliff amount, and initially unlocked amount;
- The map of recipient vestings ( `RECIPIENT_VESTINGS` ) should appropriately map the address of the recipients to all their vesting IDs;
- A vesting operation cannot be claimed if it is in its timelock period;
- When creating vesting operations using the batch option (calling `create_vesting_batch` , the length of all elements in the structure holding the list of parameters ( `create_vesting_batch_params` ) should be the same;
- The function reserved for withdrawing the full balance of tokens ( `withdraw_other_token` ) must not be capable of withdrawing the vesting token;

- 
- Only the administrators of the Token Vesting Manager can create, revoke, and fully withdraw arbitrary, non-vesting tokens;
  - Only the recipients of vesting operations can claim the vested amounts.

## Further invariants identified in the audit engagement

---

- Considering the vesting token, any token amount exceeding the number of tokens reserved for vesting can only be claimed by administrators;
- Excluding the vesting token, all tokens held by the token vesting manager can only be withdrawn by the admin;
- Administrative tasks can only be performed by addresses in the ADMINS map, and that map entry should be set to true (representing an active, authorized administrator);
- The initialization function cannot be executed successfully twice;
- Administrative values can only be overridden by addresses that have administrative (or ownership, when applied) status in the protocol;
- The number of active administrators `ADMIN_COUNT` is always equal to the number of members of the `ADMINS` maps with their administrative status set to true.



## Findings

---

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

All findings have a severity level and an execution difficulty level, ranging from low to high, as well as categories in which it fits. For more information about the classifications of the findings, refer to our [Smart Contract Analysis](#) page (adaptations performed where applicable).

# [A1] Integer Overflow Possibility When Calculating Claimable Amounts

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

## Description

There's a potential overflow issue in the `calculate_vested_amount` function, specifically in the multiplication step used to determine the number of vested tokens at a given time:

```
let linear_vest_amount: i128 = (vesting.linear_vest_amount
    * truncated_current_vesting_duration_secs)
    / final_vesting_duration_secs;
```

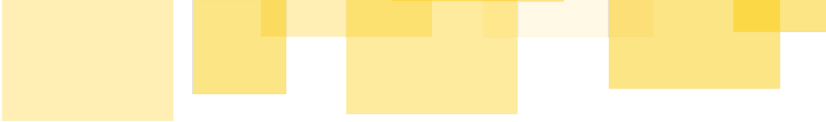
If `linear_vest_amount` and `truncated_current_vesting_duration_secs` are large enough, this can result in an overflow, which could prevent recipients from claiming their vested tokens. While this might be unlikely in practice, unless strict upper limits are enforced on these values, it's best to avoid directly multiplying unbounded numbers.

Different factors may also increase the possibility of this issue being triggered, such as the number of decimal places configured for the vesting token and properties such as those mentioned in finding [\[B3\] Vesting operations can start in the past](#).

We also highlight that this situation cannot be reverted once it is in a state that can be triggered. All options to remove the vested tokens from the contract will execute the code section in which the overflow happens, and the vested tokens of this vesting operation will remain in the contract indefinitely.

This issue was identified using the [Komet testing suite](#). We developed a property-based test, `test_all_vested_at_the_end`, which verifies that the full vesting amount is released by the end of the schedule.

```
pub fn test_all_vested_at_the_end(
    env: Env,
    start_t: u64,
    end_t: u64,
    linear_amt: i128,
    interval: u64,
) -> bool
```



The test is parameterized with `start_t` , `end_t` , `linear_amt` , and `interval` . These parameters allow us to create arbitrary vesting schedules, verifying that the `calculate_vested_amount` function works as expected for a wide range of scenarios. We used Komet in fuzzing mode to generate test cases with varying values for these parameters, uncovering scenarios where the multiplication in `calculate_vested_amount` led to an overflow. This test, along with its findings, has been provided to the client as part of the engagement's deliverables.

---

## Recommendation

Considering the issue elaborated above, we offer two possible solutions:

- Apply constraints when creating the vesting to ensure future overflows cannot occur.
  - Reformulate the calculation to avoid overflow while preserving correctness. One common approach in mathematical operations with numbers of limited size is to prioritize division over multiplication to prevent overflows while also handling potential precision loss from integer division.
- 

## Status

This finding has been addressed by reformulating the method for calculating the claimable amounts ([PR #2](#)).



## [A2] Function for Setting Administrators Can Suffer Denial of Service

---

Severity: Medium

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

---

### Description

It is possible to cause a Denial-of-Service on the `set_admin` function of the Token Vesting Manager when attempting to revoke administrative status from an address.

The `set_admin` function receives an administrator address and a status named `is_enabled`. This status indicates whether the address has administrative capabilities, represented by `true` and `false`, respectively. An address cannot be removed once added to the map of administrators (`ADMINS`), as there are no implemented methods to do so. The only way to revoke the administrative status of a present admin is to set his entry in the `ADMINS` map to false.

Whenever an administrator is added with its admin status set to `true`, a counter that tracks the number of administrators for the Token Vesting Manager is increased. Whenever an admin status is modified to false, the counter is deducted by 1.

If you call `set_admin` providing an address that is not in the `ADMINS` map and provide the `is_enabled` parameter as `false`, the Token Vesting Manager contract adds an entry to the map of administrator addresses and deducts the admin counter by 1, as it considers the status as the only criteria for modifying the administrator counter variable. If there are two active admins in the contract when triggering a call for `set_admin` with the informed parameters, the number of entries in the map becomes 3, but the counter reflecting the number of admins becomes 1.

Additionally, the `set_admin` function has an assertion forcing the number of administrators to always be above 1. Therefore, any calls to disable an administrator will fail, even if there are two or more active administrators in the protocol.

In a scenario with multiple administrators, a compromised or malicious admin could exploit this issue to prevent their own removal, effectively blocking any attempts to revoke their administrative privileges.

---





## Recommendation

Validate if the address of the administrator being modified is in the `ADMINS` map or not. If the address is not on the map and its administrative status is `false` , do not modify the counter.

---

## Status

This finding has been addressed by enforcing that an address not present in the `ADMINS` map cannot have its `is_enabled` status set to false when added through the `set_admin` function ([PR #3](#)).



## [A3] Failure to properly execute the administrative access validations

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

### Description

This issue affects all admin-only functions in the Token Vesting Manager that rely on access control checks.

Currently, the check only verifies whether the caller exists in the `ADMINS` map but does not confirm if their value is true. If an admin is removed by setting their administrative status to `false` using the `set_admin` function, they can still pass the administrative access check.

The current implementation of the access control check is as follows:

```
if !admins.get(caller).is_some() {  
    panic!("Not an admin");  
}
```

### Recommendation

We suggest updating the check to ensure only active admins (with a true value) can pass:

```
if !admins.get(caller).unwrap_or(false) { // or use Self::is_admin  
    panic!("Not an admin");  
}
```

This guarantees that if an admin is removed (administrative status set to `false`), they can no longer access admin-only functions.

Additionally, we recommend refactoring the check into a reusable function to improve maintainability and keep the logic consistent across the contract.

### Status

This finding has been addressed by implementing an `admin_check` function, which implements the recommendation provided above ([PR #4](#)).



## [A4] Initialization of the Token Vesting Manager uses forwarded, non-verified parameters

---

Severity: Low

Difficulty: Low

Recommended Action: Fix Code

Not addressed by client

---

### Description

The parameters for the initialization of the Token Vesting Manager are provided in the `init_args` vector for the `new_token_vesting_manager` function in the Token Vesting Factory. There are no validations over the addresses provided in this vector. They are simply forwarded on a call to the `init` function of a recently deployed Token Vesting Manager.

For example, `factory_caller` should reflect the address of the individual who called the Factory contract, but it can be whatever address the caller wants to be, essentially creating a Token Vesting Manager on behalf of someone else. If the owner of the address in which the Token Vesting Manager contract has been created on behalf is not aware of its existence, this newly created manager contract becomes useless.

From our current analysis, this may not lead to any major exploits but can be used in a griefing attack or result in unpredicted behavior.

---

### Recommendation

In the Token Vesting Factory, ideally, `init_args` would be built within the `new_token_vesting_manager` function, with the caller's address and a token address provided as parameters.

---

### Status

The client has acknowledged this finding, as creating vesting operations on behalf of other users is part of the protocol's business logic.



## [A5] Vesting Tokens Can Be Unlawfully Withdrawn By Administrators

---

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

---

### Description

On the Token Vesting Manager, the `create_vesting` function parameters contain signed integer types for values representing token amounts and time. This finding highlights three variables: `initial_unlock`, `cliff_amount`, and `linear_vest_amount`. These three variables are typed `i128`.

When creating a vesting operation, a certain number of tokens are transferred to the contract, representing the sum of these three values provided as parameters. In other words, the administrator deposits a number of tokens that are supposed to be claimed by a recipient. These tokens are supposed to be locked until certain restrictions on their release time are met, yet such a combination of actions and restrictions spawns unpredicted and dangerous corner cases.

One example of such cases can be achieved by using a negative initial amount and/or cliff amount, which would result in failure to claim the operation until the vested amount surpasses  $(\text{initial\_unlock} + \text{cliff\_amount}) * -1$ , configuring a Denial-of-Service, as a user would attempt to withdraw a negative amount of tokens.

Furthermore, the worrisome scenario manifests if a vesting operation is created with a positive `initial_unlock` and `cliff_amount` while using a negative `linear_vest_amount`. If not enough time has passed to vest a percentage of the linear vested amount, the only tokens that can be claimed are the initially unlockable and the cliff amounts. These values can be greater than the calculated total amount of that specific vesting operation, but this won't prevent them from being withdrawn.

This would allow an admin to withdraw tokens of other vesting operations, regardless of whether they are vested, configuring a scenario of asset theft. Additionally, a corrupted admin can unlawfully withdraw the full balance of vesting tokens.

---



## Scenario

To exemplify this, consider a vesting operation where `initial_unlock = 10`, `cliff_amount = 10`, and `linear_vest_amount = -19`. `total_expected_amount`, which is the sum of these numbers becomes 1, and only 1 is added to the number of tokens reserved for vesting (a balance of tokens that is reserved only for recipients). As a result, the admin initiating this vesting operation proceeds to deposit 1 unit of a token to the Token Manager Contract. For this scenario, assume that the administrator is using their own address as the vesting operation recipient.

If the tokens in this vesting operation are claimed when the locked values can already be claimed but no amount of the linear vested tokens can be, the recipient will claim 20 tokens, and these 20 tokens will be deducted from the number of tokens reserved for vesting. This would mean that tokens from other vesting operations are being withdrawn, which would constitute asset theft.

Simply put, a corrupted administrator would have deposited 1 token and withdrawn 20.

---

## Recommendations

Do not allow negative values for time and token amounts, especially when provided as parameters. For this specific case, use unsigned integers instead of signed.

---

## Status

This finding has been addressed by introducing checks enforcing that `initial_unlock`, `cliff_amount`, and `linear_vest_amount` are greater or equal to 0 ([PR #5](#) and [PR #6](#)).



## Informative Findings

---

The findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need external support or deviate from best practices. We have also included information on potential code size reductions and remarks on the operational perspective of the contract.

# [B1] Best practices recommendations

Severity: Informative

Recommended Action: Fix Code

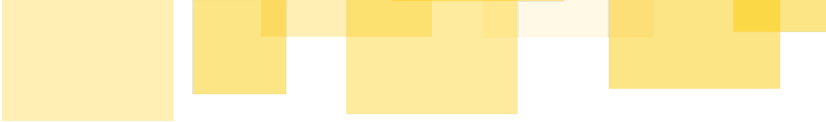
Partially addressed by client

## Description

Here are some notes on the protocol particularities, comments, and suggestions to improve the code or the business logic of the protocol in a best-practice sense. They do not present issues with the audited protocol themselves. Still, they are advised to either be aware of or to follow when possible, and they may explain minor unexpected behaviors in the deployed project.

1. `create_vesting_internal` duplicates most of the code in `create_vesting`. This adds 116 lines to the Token Vesting Manager contract (including blank lines);
2. In the function for atomically creating a batch of vesting operations on the Token Vesting Manager, `CreateVestingBatchParams` is a struct that contains several vectors. Each vector holds a specific parameter type for the `create_vesting_internal` function. This leads to additional efforts to ensure, for instance, that all vectors in this structure have the same length.
3. In some parts of the code, new elements are added to the end of a vector using insert with the vector's length ( `recipients.insert(recipients.len(), recipient.clone());` );
4. In some sections of the contract code, `unwrap_or` is used with costly default values when fetching data from storage (e.g., `_.unwrap_or(Map::new(&env))` ). This can lead to inefficiencies because the default value is always evaluated, even if the storage already contains the requested value. Since `Map::new` is a host function, it leaves the Wasm VM to create a new map object, which can be expensive;
5. In different sections of the contracts' code, the `unwrap` method is misused. `unwrap` is commonly employed for error handling, retrieving the inner value of an Option or Result. If an error or None occurs, it triggers a panic and crashes the program;
6. Using an older version of Soroban can be dangerous, as it may have bugs or security issues;
7. In different sections of the contracts' code, changes in storage are performed, and no events informing of this are triggered;

## Recommendations



For each of the topics elaborated above, we recommend implementing the following approaches into the protocol's contracts:

1. Modify the code of `create_vesting` to do the authentication checks and then call `create_vesting_internal` ;
2. Instead of using `CreateVestingBatchParams` , a structure of vectors, have a single vector of a structure containing the individual parameters for `create_vesting_internal` ;
3. When adding elements to the end of a vector, use the `push_back` method, making the code cleaner and more intuitive;
4. To avoid unnecessary calls to the host function, use `unwrap_or_else` so that `Map::new(&env)` is only invoked when needed;
5. Before `unwrap` ing values, validate that it is possible to do so safely;
6. Use the latest Soroban version available;
7. Emitting an event when storage changes is a good practice to make the contracts more transparent and usable to its clients and observers;

---

## Status

The following topics of this finding have been addressed: 1 ([PR #6](#)), 3 ([PR #7](#)), 4 ([PR #8](#)), 6 ([#PR 10](#)), and 7 ([PR #9](#)).

The remaining findings have been acknowledged and considered part of the business logic or non-dangerous to the protocol's health.





## [B2] Misusage of Storage

---

Severity: Informative

Recommended Action: Fix Code

Addressed by client

---

### Description

Persistent storage is being used where instance storage is more appropriate. Persistent storage incurs higher costs due to its long-term nature, impacting both gas fees and performance. Additionally, each individual entry key requires individual time-to-live (TTL) management. Instance storage is frequently used when the stored data is directly tied to the existence of the contract itself, which is the case for all data in the contracts in scope, and the data they handle does not grow indefinitely.

Furthermore, regardless of whether the storage entries are persistent or instance, to be readily available on calls from users/contracts, the data must not be archived. If expired and, consequently, archived, a `RestoreFootprintOp` operation must be submitted to reinstate the contract data and make it usable again.

---

### Recommendation

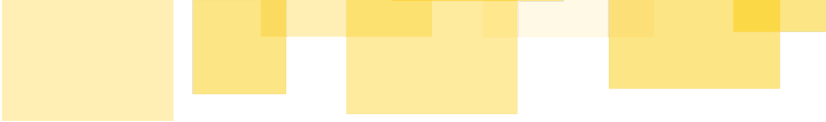
Follow the guidelines for storage usage provided in the Soroban documentation ([1],[2]), and use persistent storage for variable, non-bounded data while keeping core, fixed-sized data on instance storage.

Furthermore, to avoid needing to manually reinstate the contract data after properly modifying the storage entries from persistent to instance, we recommend extending the time-to-live of the contract data within the contract itself whenever operations concerning the data are performed. For example, in the Token Vesting Factory contract, whenever calling the `new_token_vesting_manager` function, make the first operation performed by that function extend the instance storage time-to-live.

For persistent data, extend the TTL of the specific storage keys whenever an operation that requires its usage is executed.

---

### Status



This finding has been addressed by the client by following the recommendation above ([PR #12](#)).



## [B3] Vesting operations can start in the past

---

Severity: Informative

Recommended Action: Document Prominently

Not addressed by client

---

### Description

Currently, the only validation for preventing vesting operations from starting in the past is ensuring that the start time UTC timestamp is nonzero (`start_timestamp != 0`). This allows users to create vesting operations starting as far back as 1970 (UTC timestamp).

This can greatly increase the values used in the calculation for the claimable amount in vesting operations, specifically the `current_vesting_duration_secs` variable. By combining this in a scenario that uses a token with a high number of decimals, the overflow issue elaborated in finding [\[A1\] Integer Overflow Possibility When Calculating Claimable Amounts](#) will be more likely to be triggered.

---

### Recommendation

Have validations enforcing that vesting operations can only start after a certain point, be it the current ledger timestamp or at least a couple of months back (or since the start of your platform) if allowing users to create vesting operations with `start_time` in the past.

---

### Status

The client has acknowledged this finding, as traditional, paper-based, or web2 digital vesting operations could be transferred to the blockchain. Therefore, this is considered part of the protocol's business logic.

# [B4] Token contract calls are performed using `env.invoke_contract`

Severity: Informative

Recommended Action: Fix Code

Addressed by client

## Description

When handling tokens in the Token Vesting Manager, token contract calls are performed using raw contract calls via `env.invoke_contract` :

```
let _ : Val = env.invoke_contract(  
    &token_address,  
    &Symbol::new(&env, "transfer_from"),  
    vec![  
        &env,  
        env.current_contract_address().to_val(),  
        caller.to_val(),  
        env.current_contract_address().to_val(),  
        total_expected_amount.into_val(&env),  
    ],  
);
```

This approach requires manually constructing the invocation, which can be error-prone and harder to read.

## Recommendation

To improve readability and maintainability, consider using the SDK's `TokenClient` wrapper for token contracts.

```
TokenClient::new(&env, &token_address)  
    .transfer_from(  
        &env.current_contract_address(),  
        &caller,  
        &env.current_contract_address(),  
        &total_expected_amount,  
    );
```

## Status

This finding has been addressed by the client by following the recommendation above ([PR #11](#)).



## [B5] The Creation of Vesting Operations in Batch May Fail for Gas-Related Reasons

---

Severity: Informative

Recommended Action: Fix Code

Not addressed by client

---

### Description

The Token Vesting Manager enables its users to create several vesting operations at once through its `create_vesting_batch` endpoint. Users can call this endpoint while providing all parameters for each of the vesting operations in vectors within a single `struct`. The Token Vesting Manager iterates through the vectors of parameters and performs the calls for the internal function that triggers the creation of vesting operations.

The concerning detail within this `create_vesting_batch` function is the presence of an unbounded loop, used to iterate through the vesting operations parameters. Due to blockchain resource limitations, the user might not be able to have their transaction successfully processed if the number of vesting operations to be created is too great.

---

### Recommendation

Impose a reasonable limit on the number of vesting operations that can be processed in a single transaction to ensure the loop remains bounded and transactions execute reliably.

---

### Status

The client has acknowledged this finding as it is not capable of causing any particular damage to the platform's users.