Zellic

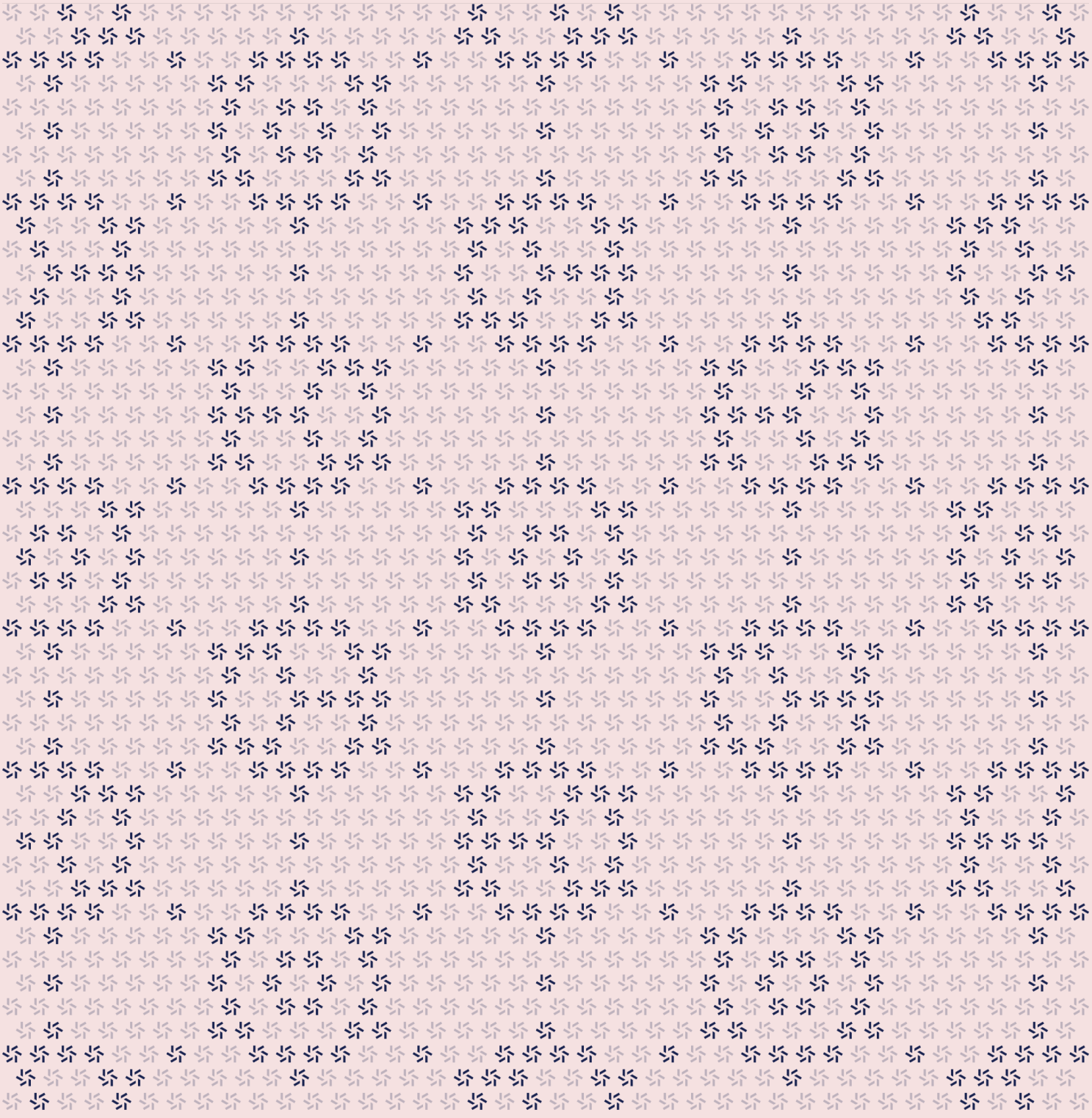
Prepared for
Tom Rogers
Sentora

Prepared by
Ethan Lee
Ayaz Mammadov
Zellic

December 16, 2025

Stellar Vault

Stellar Application Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Sentora from December 5th to December 12th, 2025. During this engagement, Zellic reviewed Stellar Vault's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an attacker withdraw more tokens than deposited from the vault?
- Could arithmetic overflow/underflow cause incorrect balance calculations?
- Could unauthorized users bypass access control and call admin functions?
- Could a malicious user claim rewards multiple times using the same Merkle proof?
- Could an attacker forge or manipulate Merkle proofs to claim unauthorized rewards?
- Could incorrect Merkle-root updates allow invalid claims?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Setting up the reward program and generating Merkle proofs off-chain

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

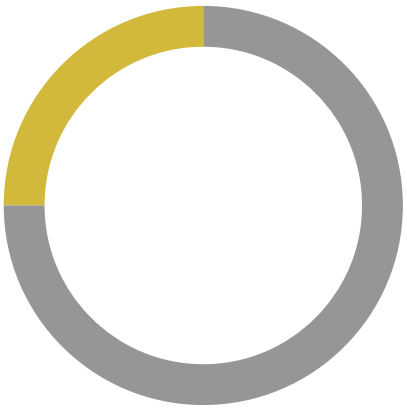
During our assessment on the scoped Stellar Vault files, we discovered four findings. No critical issues were found. One finding was of medium impact and the other findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of

Sentora in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	0
Informational	3



2. Introduction

2.1. About Stellar Vault

Sentora contributed the following description of Stellar Vault:

> Sentora is a platform that bridges traditional finance and decentralized finance by providing secure, institutional-grade access to DeFi opportunities. Designed for institutional investors, Sentora enables seamless participation in yield generation, asset management, and other advanced DeFi strategies through an intelligent and unified interface.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the files.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped files itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Stellar Vault Files

Type	Rust
Platform	Stellar
Target	StellarVaults
Repository	https://github.com/Into-The-Block-Corp/StellarVaults ↗
Version	`b4b61b46273eb66855777d405b485150f29cb484`
Programs	`vault/src/**.rs` `escrow/src/**.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 0.9 person-weeks. The assessment was conducted by two consultants over the course of six calendar days.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

The following consultants were engaged to conduct the assessment:

Ethan Lee
Engineer
ethl@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 5, 2025	Kick-off call
December 5, 2025	Start of primary review period
December 12, 2025	End of primary review period

3. Detailed Findings

3.1. Insufficient test coverage

Target	Project-wide		
Category	Protocol Risks	**Severity**	Medium
Likelihood	N/A	**Impact**	Medium

Description

When building a complex contract with multiple moving parts (state changes) and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios on every function that touches the contract's state.

Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, and all functions — not just surface-level functions. It is important to test the invariants required for ensuring security and also verify any mathematical properties from the project's specification. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

Unit test cases that are missing include (but are not necessarily limited to) those documented in section 5. ↗.

Impact

Good test coverage has multiple effects:

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in the product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point may seem contradictory given the time investment to create and maintain tests. However, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks other code if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence.

Tests have your back here. They are an excellent indicator that the existing functionality was most likely not broken by a change to the code. Without a comprehensive test suite, the above benefits

are lost. This increases the likelihood of bugs and vulnerabilities going unnoticed until after deployment, which can be costly and damaging to the project's reputation.

Recommendations

We recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Remediation

This issue has been acknowledged by Sentora, and a fix was implemented in commit 79139687 ↗.

3.2. Inconsistent casting to `i128`

Target	Project-wide		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

Many functions accept `u128` as a parameter, though this is often cast into `i128`. Sometimes this is done using `as i128` and other times with `i128::try_from`.

```
fn claim(e: Env, from: Address, vault: Address, epoch: u32, claim:
    ClaimParams) -> Result<(), ContractErrors> {
    ...
    let amount_i128 = i128::try_from(amount).map_err(|_|
    ContractErrors::RewardAmountTooLarge)?;
}
```

```
pub fn handle_deposit(e: &Env, from: Address, amount: u128) -> u64 {
    ...
    token_client.transfer(&from, &contract_address, &(amount as i128));
    ...
}
```

Impact

As of the time of writing, there are no security issues with the logic as in most cases where `as i128` is used, it is used for `try_transfer` which internally validates the value supplied. The inconsistent casting approach does not produce clear error messages, which may cause confusion for users and make debugging difficult.

Recommendations

If a cast is done, prefer to use `i128::try_from`. If a decision is made to do an unchecked cast, then document it; in this case, the solution would be documenting the unsafe cast because the `transfer` function itself does the relevant checks.

Remediation

This issue has been acknowledged by Sentora, and a fix was implemented in commit 79139687 ↗.

3.3. Checks-effects-interactions pattern unaccounted for

Target	Project-wide		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In various areas of the project, the checks-effects-interactions pattern is not taken into account. Specifically, the interaction should happen after the effect. See two examples below. This affects both the escrow and the vault contracts.

```
fn withdraw(e: Env, from: Address, amount: u128) -> Result<(),
    ContractErrors> {
    ...
    withdraw_deposit_asset(&e, &from, &amount)?;
    reduce_total_principal(&e, &amount);
    bump_instance(&e);
    ...
}
```

```
fn claim(e: Env, from: Address, vault: Address, epoch: u32, claim:
    ClaimParams) -> Result<(), ContractErrors> {
    ...
    token_client.transfer(&contract_address, &from, &amount_i128);
    set_claimed(&e, &vault, epoch, deposit_id);
    ...
}
```

Impact

Stellar Soroban does not have contract reentrancy as of the time of writing. However, following this pattern would offer an extra layer of defense against issues that may potentially arise in the future.

Recommendations

Adjust the order of operations to correctly account for the checks-effects-interactions pattern.

Remediation

This issue has been acknowledged by Sentora.

3.4. Combined getter-setter functionality

Target	Project-wide		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There is a pattern used by this codebase in which a method acts as both a getter and a setter depending on the value inside the `Option<Address>`.

```
pub fn admin(e: &Env, value: Option<Address>) -> Option<Address> {
    if let Some(v) = value {
        e.storage().instance().set(&StorageKeys::Admin, &v);
    }

    e.storage().instance().get(&StorageKeys::Admin)
}
```

Impact

This can be confusing as any incorrect invocation or incorrect argument supplied to `admin` could result in a value being set, when that was not the intention.

Specifically, for the `admin` getter/setter, it is only ever called with a value in the constructor. However, for other getters/setters, this is more applicable as they are set during the contract's lifetime.

Recommendations

Split the functionality into separate methods.

Remediation

This issue has been acknowledged by Sentora.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Merkle-tree domain separation

The Escrow contract is responsible for verifying claims. This is done by taking claim information alongside a Merkle proof; these are used to compute a Merkle root that is then checked against a stored root. That specific leaf is then marked as `claimed` such that it cannot be reused.

```
fn claim(e: Env, from: Address, vault: Address, epoch: u32, claim:
    ClaimParams) -> Result<(), ContractErrors> {
    ...
    let leaf = compute_leaf_hash(&e, &vault, epoch, deposit_id, &from, amount);
    let computed_root = compute_root_from_proof(&e, &leaf, &proof,
    leaf_index);

    if computed_root != epoch_data.root {
        return Err(ContractErrors::RewardInvalidProof);
    }
    ...
}
```

To prevent the second preimage attack, the Merkle leaf is not supplied but instead created from the leaf inputs. The inputs of the leaf to the hash function are prefixed with a fixed domain separator.

```
pub fn compute_leaf_hash(e: &Env, vault: &Address, epoch: u32, deposit_id:
    u64, owner: &Address, amount: u128) -> BytesN<32> {
    let mut payload = Bytes::from_slice(e, REWARD_LEAF_PREFIX);
    let encoded = (vault.clone(), epoch, deposit_id, owner.clone(),
    amount).to_xdr(e);
    payload.append(&encoded);
    e.crypto().sha256(&payload).into()
}
```

However, when creating the Merkle proof, the hashes themselves are not prefixed with a domain separator — though, in this case, due to the different hash-input lengths for nodes and leaves, this is not a security issue. To be consistent with other implementations, both the leaves and the nodes should have different prefixes.

```
pub fn compute_root_from_proof(e: &Env, leaf: &BytesN<32>, proof:
    &soroban_sdk::Vec<BytesN<32>>, mut index: u32) -> BytesN<32> {
```

```
    let mut hash = leaf.clone();
    ...

        if index % 2 == 0 {
            data[..32].copy_from_slice(&current_bytes);
            data[32..].copy_from_slice(&sibling_bytes);
        } else {
            data[..32].copy_from_slice(&sibling_bytes);
            data[32..].copy_from_slice(&current_bytes);
        }

        let buffer = Bytes::from_slice(e, &data);
        hash = e.crypto().sha256(&buffer).into();
        index /= 2;
    ...
    }
```

4.2. Escrow centralization

As of the time of writing, there is no system that ensures that funds deposited into the vault are guaranteed to be part of the Merkle root that is set in the Escrow contract — the Merkle root is calculated offchain from on-chain transactions that are stored. Neither is there a guarantee that funds deposited in the vault will be sent back to the escrow. If the escrow does not have enough funds, it will error out during a user's claim. It is the responsibility of the team to ensure that funds/rewards collected from reward deposits are returned to the escrow.

It is recommended to generate proper Merkle proofs and verify through sufficient testing that the contract has adequate funds to fulfill claims.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: Vault

Description

The Vault contract is a custodial vault that allows users to deposit and withdraw a single token asset. Users can deposit tokens with an optional referral ID, which transfers tokens to the contract and generates a unique deposit ID. Admin functions include contract upgrade, admin address changes, and pause/unpause controls for deposit operations. This is a pure custody vault with no staking, rewards, or timelock mechanisms.

Storage

- `CoreStorageKeys::Admin`
  - Type: Instance.
  - TTL extensions: Extended by `bump_instance()` called in `upgrade()`, `update_admin()`, `set_status()`, `deposit()`, and `withdraw()`.
  - If expired: Entire instance storage becomes archived.
- `CoreStorageKeys::DepositAsset`
  - Type: Instance.
  - TTL extensions: Extended by `bump_instance()`.
  - If expired: Entire instance storage becomes archived.
- `CoreStorageKeys::Paused`
  - Type: Instance.
  - TTL extensions: Extended by `bump_instance()`.
  - If expired: Entire instance storage becomes archived.
- `DepositStorageKey::NextDepositId`
  - Type: Instance.
  - TTL extensions: Extended by `bump_instance()` called in `deposit()`.
  - If expired: Entire instance storage becomes archived.
- `DepositStorageKey::TotalPrincipal`
  - Type: Instance.
  - TTL extensions: Extended by `bump_instance()` called in `deposit()` and

`withdraw()`.
- If expired: Entire instance storage becomes archived.

- `DepositStorageKey::Deposit`
  - Type: Persistent.
  - TTL extensions: Extended by `deposit()` when the value is updated.
  - If expired: User's deposit record becomes archived.

Test coverage

Additional tests were added during the remediation period. The following section reflects the test coverage at the time of the initial review.

Cases covered

- `update_admin()`
  - Update the admin successfully with proper authorization.
  - Revert when the admin has not signed the transaction.
- `set_status()`
  - Set status to true/false successfully with proper admin authorization.
  - Revert when the admin has not signed the transaction.
- `deposit()`
  - Deposit successfully with event emission and state update.
  - Emit `referral_id` in the event when provided.
  - Revert with the `VaultPaused` error when the vault is paused.
- `withdraw()`
  - Withdraw the full deposit amount successfully.
  - Perform partial and multiple withdrawals successfully.
  - Revert when the user has not signed the transaction.
  - Revert with `NotEnoughDeposit` when no deposit exists.
  - Revert with `NotEnoughDeposit` when the amount exceeds the deposit.

Cases not covered

- `upgrade()`
  - No test for a successful upgrade with proper admin authorization.
  - No test for reverting when a non-admin user calls upgrade.
  - No test for panicking when the admin is `None`.
- `update_admin()`
  - No test for behavior when the admin storage is `None`.
- `set_status()`

    - No test for panicking when the admin is `None`.
- `deposit()`
    - No test for reverting when the depositor has not signed the transaction.
    - No test for reverting when the token balance is insufficient for transfer.
    - No test for behavior when the amount is zero.
- `withdraw()`
    - No test for the `WithdrawDepositAssetFailed` error when token transfer fails.

Attack surface

- The `withdraw()` function has no pause check while `deposit()` does, so users can withdraw even when the vault is paused.
- Casting `u128` to `i128` in token transfers can overflow for values exceeding the `i128` maximum.
- Deposit and withdraw properly require user authorization; `upgrade()` and `set_status()` safely panic when the admin is `None`.

These surfaces are not exploitable.

5.2. Component: Escrow

Description

The Escrow contract is a Merkle-tree–based reward-distribution system. An admin can set reward roots for each vault's epoch, which includes asset address, total rewards, leaf count, and program end timestamp. Users can claim their rewards by providing a valid Merkle proof that verifies their deposit ID, owner address, and reward amount. Admin functions include contract upgrade and withdrawal of funds from the contract. The contract uses bitmap-based tracking for claimed rewards to prevent double-claiming.

Storage

- `StorageKeys::Admin`
    - Type: Instance.
    - TTL extensions: Extended by `bump_instance()` called in `upgrade()`, `withdraw()`, `set_rewards_root()`, and `claim()`.
    - If expired: Entire instance storage becomes archived.
- `StorageKeys::LatestRewardEpoch`
    - Type: Instance.
    - TTL extensions: Extended by `set_latest_epoch()` in `set_rewards_root()`.

    - If expired: Entire instance storage becomes archived.
- `StorageKeys::RewardEpoch`
    - Type: Persistent.
    - TTL extensions: Extended by `put_reward_epoch()` in `set_rewards_root()` and by `bump_reward_epoch()` in `claim()`.
    - If expired: Reward epoch data becomes archived. Claims for this epoch will fail.
- `StorageKeys::RewardClaimed`
    - Type: Persistent.
    - TTL extensions: Extended by `set_claimed()` when the claim is recorded.
    - If expired: Claimed status becomes archived.

Test coverage

Additional tests were added during the remediation period. The following section reflects the test coverage at the time of the initial review.

Cases covered

- `withdraw()`
    - Withdraw multiple tokens successfully with proper admin authorization.
    - Revert when the admin has not signed the transaction.
- `set_rewards_root()`
    - Set rewards root successfully with event emission and state update.
    - Revert with the `RewardEpochNotMatured` error when `program_end_ts` is in the future.
- `claim()`
    - Claim rewards successfully with a valid Merkle proof.
    - Revert with the `RewardLeafAlreadyClaimed` error on a duplicate claim attempt.
    - Revert with the `RewardInvalidProof` error when the amount does not match the proof.
    - Revert with the `RewardEpochNotFound` error when the epoch does not exist.

Cases not covered

- `upgrade()`
    - No test for a successful upgrade with proper admin authorization.
    - No test for reverting when a non-admin user calls upgrade.
    - No test for panicking when admin is `None`.
- `withdraw()`

- No test for a panic when the admin is `None`.
- No test for overflow when the amount exceeds the `i128` maximum.

- `set_rewards_root()`
  - No test for the `RewardEpochOutOfOrder` error when the epoch is not strictly increasing.
  - No test for the `InsufficientEscrowBalance` error when the contract balance is insufficient.
  - No test for the `RewardAmountTooLarge` error when `total_rewards` exceeds the `i128` maximum.
  - No test for panicking when the admin is `None`.
- `claim()`
  - No test for reverting when the claimer has not signed the transaction.
  - No test for the `RewardEpochNotMatured` error when claiming before `program_end_ts`.
  - No test for the `RewardLeafIndexOutOfBounds` error when `leaf_index >= leaf_count`.
  - No test for the `RewardInvalidProof` error when the amount is zero.
  - No test for the `RewardAmountTooLarge` error when the claim amount exceeds the `i128` maximum.
  - No test for the `InsufficientEscrowBalance` error when the contract balance is insufficient for the claim.

Attack surface

- Casting `u128` to `i128` in the withdraw function can overflow for values exceeding the `i128` maximum, potentially causing unexpected transfer amounts. This contract has overflow checks by configuration to revert on overflow.
- Admin functions (`upgrade()`, `withdraw()`, `set_rewards_root()`) should panic when the admin is `None`.
- The withdraw function transfers `balance - amount` to the admin, which could underflow if the amount is greater than the balance.

These surfaces are not exploitable.

6. Assessment Results

During our assessment on the scoped Stellar Vault files, we discovered four findings. No critical issues were found. One finding was of medium impact and the other findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.