Zellic

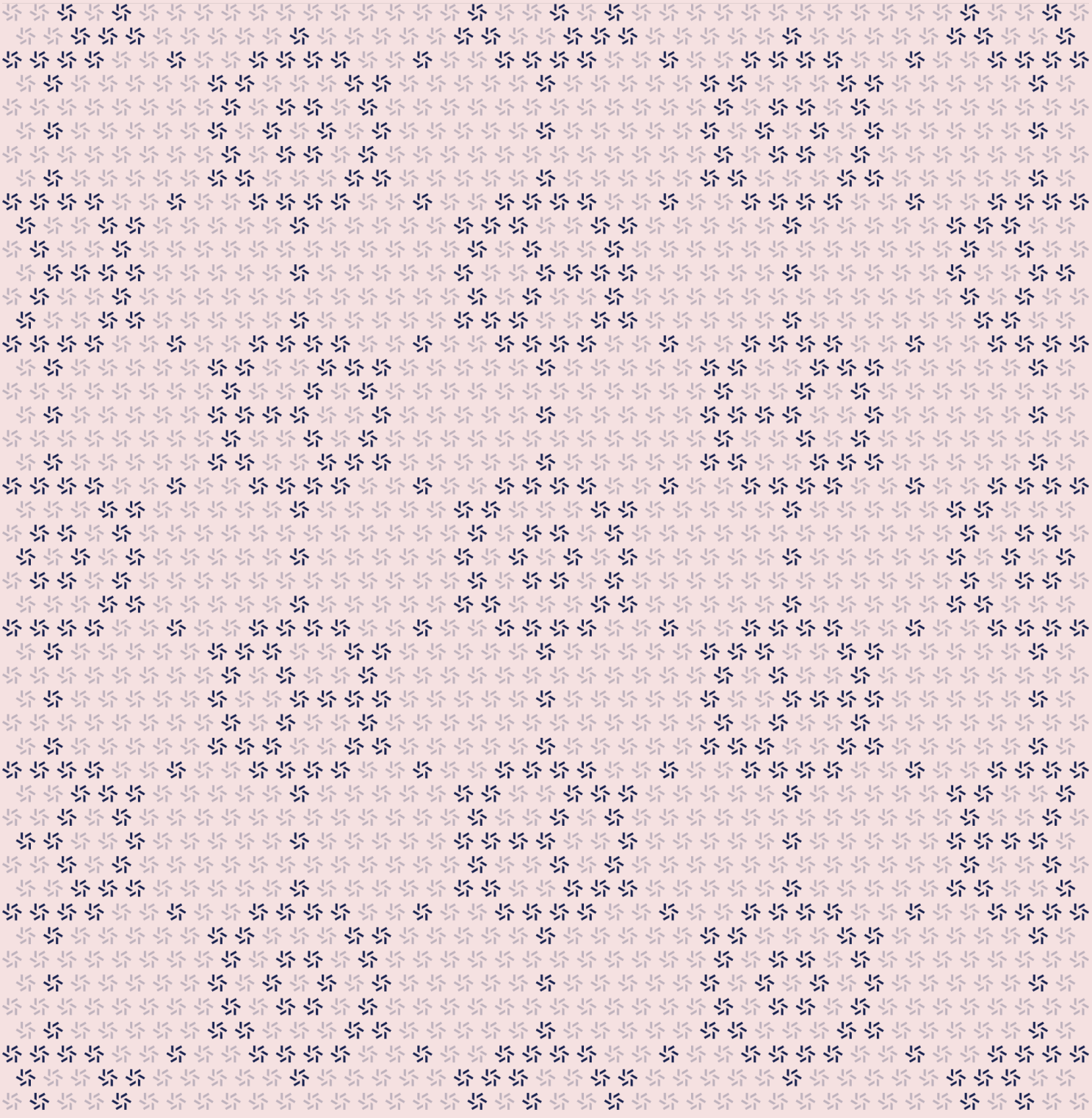
Prepared for
Tom Rogers
Sentora

Prepared by
Qingying Jie
Sihoon Lee
Zellic

May 12, 2026

Pre-deposit Program

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **5**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **7**

2.1. About Pre-deposit Program 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **11**

3.1. Centralization risk 11

3.2. Unbounded expired epoch sweep 13

3.3. Unbounded withdrawal 15

3.4. Insufficient test coverage 17

3.5. The function `sweep_expired_epoch` incorrectly assumes that expired persistent storage entries will be automatically removed 19

3.6. Withdrawal silently skips assets below retention amount 21

3.7. Updating the admin might allow unauthenticated initialization path 23

3.8. Latest epoch TTL is shorter than reward-epoch TTL 25

3.9. Admin fund recovery lacks event emission 27

4. Discussion 29

4.1. Deployment verification 29

5. System Design 30

5.1. Component: Escrow 30

5.2. Component: Vault 31

6. Assessment Results 32

6.1. Disclaimer 32

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 May 7th to May 11th, 2026. During this engagement, Zellic reviewed Pre-deposit Program'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:

- Are there any vulnerabilities that could result in the loss of user funds?
- Could rewards be claimed more than once?
- Is access control of the functions implemented appropriately?

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
- Operational risks

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 Pre-deposit Program files, we discovered nine findings. No critical issues were found. One finding was of high impact, three were of medium impact, one was of low impact, and the remaining 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	1
Medium	3
Low	1
Informational	4

2. Introduction

2.1. About Pre-deposit Program

Sentora contributed the following description of Pre-deposit Program:

> The Sentora smart-contracts workspace is a Soroban (Stellar) pre-deposit program: users deposit a configured asset into the vault to accrue rewards over the program's lifetime.

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:

Pre-deposit Program Files

Type	Rust
Platform	Stellar
Target	Only changes between 7913968...a7955fc
Repository	https://github.com/Into-The-Block-Corp/StellarVaults ↗
Version	`a7955fcb72d7eb482a84d0d1cdc62c55749d3d9b`
Programs	`contracts/*.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.5 person-days. The assessment was conducted by two consultants over the course of three 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:

Qingying Jie
Engineer
qingying@zellic.io

Sihoon Lee
Engineer
sihoon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 7, 2026	Kick-off call
May 7, 2026	Start of primary review period
May 11, 2026	End of primary review period

3. Detailed Findings

3.1. Centralization risk

Target	escrow		
Category	Code Maturity	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

The escrow contract gives the admin privileged withdrawal paths that can move escrowed reward assets out of the contract; `withdraw` lets the admin provide a list of assets and a retained amount for each asset. The function then transfers the full balance above that retained amount to the admin without checking the amount of rewards already committed to users.

```
// contracts/escrow/src/contract.rs#L85-L99
fn withdraw(e: Env, actions: Vec<(Address, u128)>) {
    let admin: Address = admin(&e, None).unwrap();
    admin.require_auth();
    for (asset, amount) in actions {
        let client: token::TokenClient = token::TokenClient::new(&e, &asset);
        let balance: i128 = client.balance(&e.current_contract_address());
        let amount_i128 = i128::try_from(amount)
            .unwrap_or_else(|_| panic_with_error!(e,
    ContractErrors::WithdrawAmountTooLarge));
        let transfer_amount = balance - amount_i128;
        if transfer_amount <= 0 {
            continue;
        }
        client.transfer(&e.current_contract_address(), &admin,
    &transfer_amount);
    }
    // [...]
}
```

The same trust assumption applies to `sweep_expired_epoch` (see Finding 3.2. ↗). After an epoch's reward metadata is no longer available, the admin chooses the asset and amount to transfer. The function only checks that the contract has enough token balance then reduces committed rewards for the selected asset.

These paths make the admin a single point of control over the escrowed funds. A compromised or malicious admin can remove reward assets from the contract and prevent users from claiming rewards that were previously published through reward roots.

Impact

The above introduces a centralization risk that users should be aware of, as it grants a single point of control over the system.

Recommendations

We recommend that this centralization risk be clearly documented for users so that they are aware of the extent of the admin's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the admin may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for admin access.

Remediation

This issue has been acknowledged by Sentora.

3.2. Unbounded expired epoch sweep

Target	escrow		
Category	Business Logic	**Severity**	Critical
Likelihood	Low	**Impact**	Medium

Description

The `sweep_expired_epoch` function lets the admin choose an arbitrary `amount` for the provided `asset` after the reward metadata for the expired epoch in the vault is no longer available. The function only checks that the requested amount does not exceed the escrow contract's current token balance. It does not bind the sweep amount to the expired epoch's original `total_rewards`, remaining unclaimed rewards, or the reward asset originally configured for that epoch.

```
// contracts/escrow/src/contract.rs#L235-L262
fn sweep_expired_epoch(e: Env, asset: Address, vault: Address, epoch: u32,
    amount: u128) -> Result<(), ContractErrors> {
    let admin_addr = admin(&e, None).unwrap();
    admin_addr.require_auth();

    // Only allow sweep if the epoch data has expired (no longer in storage)
    if get_reward_epoch(&e, &vault, epoch).is_some() {
        return Err(ContractErrors::RewardEpochNotExpired);
    }

    // Prove the epoch actually existed at some point: it must be at or
    // before the latest published epoch for this vault. Without this,
    // any fabricated or future (vault, epoch) pair would pass the
    // is_none() check above.
    let latest = get_latest_epoch(&e,
    &vault).ok_or(ContractErrors::RewardEpochNotFound)?;
    if epoch > latest {
        return Err(ContractErrors::RewardEpochNotFound);
    }

    let amount_i128 = i128::try_from(amount).map_err(|_|
    ContractErrors::RewardAmountTooLarge)?;
    let token_client = token::TokenClient::new(&e, &asset);
    let balance = token_client.balance(&e.current_contract_address());

    if balance < amount_i128 {
        return Err(ContractErrors::InsufficientEscrowBalance);
```

```
    }

    token_client.transfer(&e.current_contract_address(), &admin_addr,
    &amount_i128);
    reduce_committed_rewards(&e, &asset, amount);
    // [...]
}
```

The existence check also allows epoch gaps. If `latest` is greater than the requested epoch but no reward root was ever set for that exact epoch, `get_reward_epoch` returns `None` and the sweep can proceed.

Impact

An admin can sweep more than the unclaimed rewards of the expired epoch and withdraw funds reserved for other active epochs that use the same asset. This also reduces `committed_rewards` for the asset, which can make the accounting state understate outstanding reward obligations.

Additionally, this function is centralized and can be used to drain funds from the escrow contract.

Recommendations

Consider limiting the amount to sweep to the remaining amount that has not been claimed or swept of the expired epoch and rejecting epochs that were never explicitly registered.

Remediation

This issue has been acknowledged by Sentora.

3.3. Unbounded withdrawal

Target	escrow		
Category	Business Logic	**Severity**	Critical
Likelihood	Low	**Impact**	Medium

Description

The `withdraw` function serves as a method for the admin to directly withdraw any amount of any asset from the contract. The function only checks that the requested amount does not exceed the contract's current balance for each asset.

```
fn withdraw(e: Env, actions: Vec<(Address, u128)>) {
    let admin: Address = admin(&e, None).unwrap();
    admin.require_auth();
    for (asset, amount) in actions {
        let client: token::TokenClient = token::TokenClient::new(&e, &asset);
        let balance: i128 = client.balance(&e.current_contract_address());
        let amount_i128 = i128::try_from(amount)
            .unwrap_or_else(|_| panic_with_error!(e,
    ContractErrors::WithdrawAmountTooLarge));
        let transfer_amount = balance - amount_i128;
        if transfer_amount <= 0 {
            continue;
        }
        client.transfer(&e.current_contract_address(), &admin,
    &transfer_amount);
    }
    bump_instance(&e);
}
```

Impact

This function does not impose any restrictions on withdrawals of reward assets, while the corresponding `committed_rewards` records the total amount of rewards to be claimed by users.

If the admin accidentally or maliciously withdraws reward assets, the contract balance of a specific asset could fall below its corresponding `committed_rewards`, potentially preventing users from claiming rewards for a period of time.

Recommendations

Consider restricting reward-asset withdrawals such that the remaining contract balance cannot fall below the corresponding `committed_rewards`.

Remediation

This issue has been acknowledged by Sentora.

3.4. Insufficient test coverage

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

Description

The test suite at the time of writing does not fully cover security-relevant behaviors and state transitions of the system. While core functionality has some tests, several areas remain undertested, making it difficult to confirm correctness under edge cases, failure conditions, and adversarial scenarios.

Specifically, the following areas require additional coverage:

Missing coverage generally falls into these broad categories:

- Contracts or modules with limited tests — new functions such as `sweep_expired_epoch` and `update_admin` lack sufficient tests to validate core functionality.
- Untested invariants or assumptions — invariant tests for committed rewards are not asserted. For example, `committed_rewards` increases when roots are set, decreases when users claim, and never understates active obligations after withdrawals or sweeps.

Impact

Insufficient test coverage increases the likelihood that

- bugs or vulnerabilities remain undetected until production,
- refactors or minor changes unintentionally break existing functionality, and
- critical security assumptions, such as invariant guarantees or permission checks, are violated silently.

Comprehensive testing improves code correctness, auditability, and developer confidence, and it reduces the risk of costly postdeployment fixes.

Recommendations

Expand the test suite to cover the gaps identified above, with emphasis on

- unit tests for new functions, covering both success and revert paths, and
- tests that assert key invariants and system properties after relevant operations.

Remediation

This issue has been acknowledged by Sentora, and a partial fix was implemented in commits `cb1c1260` ↗ and `bc747218` ↗.

The added tests cover escrow `sweep_expired_epoch` (committed-rewards failure, success paths, and sweep event payload) and escrow `withdraw` (admin-withdraw events when funds move, none when transfer amount is zero). The other tests described in this finding were not added.

3.5. The function `sweep_expired_epoch` incorrectly assumes that expired persistent storage entries will be automatically removed

Target	escrow		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The function `sweep_expired_epoch` attempts to withdraw unclaimed rewards from a specific epoch after some period of time and requires that the reward epoch entry does not exist. However, the reward epoch entry is stored in the persistent storage, which will not be automatically removed after expiration.

```
pub fn get_reward_epoch(e: &Env, vault: &Address, epoch: u32) ->
    Option<RewardEpoch> {
    let key = StorageKeys::RewardEpoch((vault.clone(), epoch));
    e.storage().persistent().get(&key)
}

fn sweep_expired_epoch(e: Env, asset: Address, vault: Address, epoch: u32,
    amount: u128) -> Result<(), ContractErrors> {
    let admin_addr = admin(&e, None).unwrap();
    admin_addr.require_auth();

    // Only allow sweep if the epoch data has expired (no longer in storage)
    if get_reward_epoch(&e, &vault, epoch).is_some() {
        return Err(ContractErrors::RewardEpochNotExpired);
    }
```

Impact

When a reward epoch entry expires, the call to the function `sweep_expired_epoch` will either fail with `INVOKE_HOST_FUNCTION_ENTRY_ARCHIVED` or the entry will be restored and the contract will revert with `RewardEpochNotExpired`. As a result, the subsequent function code is unreachable.

Recommendations

At the time of writing, Pre-deposit Program uses a single-epoch deployment model, so it is unlikely that the function `sweep_expired_epoch` will be called in practice. However, if the protocol is extended to support multiple epochs in the future, consider introducing an explicit expiration time and check it inside the function, while removing entries after they are expired and fully withdrawn.

Remediation

This issue has been acknowledged by Sentora, and a fix was implemented in commit `1686cc6a` ↗.

3.6. Withdrawal silently skips assets below retention amount

Target	escrow		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `withdraw` function treats the `amount` parameter in each action as the amount to keep in the contract. If the current balance is less than or equal to that retained amount, the function silently skips the asset and continues processing the remaining batch.

```
// contracts/escrow/src/contract.rs#L88-L97
for (asset, amount) in actions {
    let client: token::TokenClient = token::TokenClient::new(&e, &asset);
    let balance: i128 = client.balance(&e.current_contract_address());
    let amount_i128 = i128::try_from(amount)
        .unwrap_or_else(|_| panic_with_error!(e,
    ContractErrors::WithdrawAmountTooLarge));
    let transfer_amount = balance - amount_i128;
    if transfer_amount <= 0 {
        continue;
    }
    client.transfer(&e.current_contract_address(), &admin, &transfer_amount);
}
```

This behavior may be intentional for a batch operation, but the function name and parameter name do not make the retention-floor semantics clear to callers.

Impact

Callers may believe a withdrawal occurred even when no transfer was made for one or more assets. This can cause off-chain automation or operational playbooks to report a successful action without noticing that an asset was skipped.

Recommendations

We have two recommendations:

1. Ensure the function returns an error when an action transfers nothing.
2. Rename the parameter or action type to make the retention-floor semantics explicit.

Remediation

This issue has been acknowledged by Sentora.

3.7. Updating the admin might allow unauthenticated initialization path

Target	escrow		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `update_admin` function only requires authorization when an admin already exists. At the time of writing, the constructor sets the admin during deployment, so the unauthenticated branch should not be reachable in normal deployments. However, the function itself still encodes an implicit assumption that storage always contains an admin.

```
// contracts/escrow/src/contract.rs#L65-L82
fn __constructor(e: Env, new_admin: Address) {
    admin(&e, Some(new_admin));
    bump_instance(&e);
}

fn update_admin(e: Env, new_admin: Address) {
    if let Some(v) = admin(&e, None) {
        v.require_auth();
    }
    admin(&e, Some(new_admin.clone()));
    bump_instance(&e);
    ContractAdminEvent { address: new_admin }.publish(&e);
}
```

Other admin-only functions unwrap the admin value before requiring authorization. Applying the same pattern here would make the invariant explicit in code.

Impact

This does not create an expected exploit path under the current constructor flow. It weakens code clarity and could become relevant if future deployment, migration, or upgrade flows ever leave the admin unset.

Recommendations

Require the existing admin unconditionally with `admin(&e, None).unwrap().require_auth()`. If an initialization path is needed, keep it separate from the admin rotation function.

Remediation

This issue has been acknowledged by Sentora, and a fix was implemented in commit `714479f0`.

3.8. Latest epoch TTL is shorter than reward-epoch TTL

Target	escrow		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Reward-epoch entries are stored in persistent storage with a six-month maximum TTL, while the latest epoch marker stored in instance storage is only extended to one month. The latest epoch marker controls reward epoch ordering and is also used by `sweep_expired_epoch` to prove that a requested epoch is not in the future.

```
// contracts/escrow/src/storage.rs
const EPOCH_MIN_TTL: u32 = LEDGER_WEEK;
const EPOCH_MAX_TTL: u32 = LEDGER_MONTH * 6;

pub fn put_reward_epoch(e: &Env, vault: &Address, epoch: u32, value:
    &RewardEpoch) {
    let key = StorageKeys::RewardEpoch((vault.clone(), epoch));
    let storage = e.storage().persistent();
    storage.set(&key, value);
    storage.extend_ttl(&key, EPOCH_MIN_TTL, EPOCH_MAX_TTL);
}

pub fn set_latest_epoch(e: &Env, vault: &Address, epoch: u32) {
    let storage = e.storage().instance();
    let key = StorageKeys::LatestRewardEpoch(vault.clone());
    storage.set(&key, &epoch);
    storage.extend_ttl(LEDGER_WEEK, LEDGER_MONTH);
}
```

```
// contracts/escrow/src/contract.rs#L118-L126
if let Some(latest) = get_latest_epoch(&e, &vault) {
    if epoch <= latest {
        return Err(ContractErrors::RewardEpochOutOfOrder);
    }
}

if get_reward_epoch(&e, &vault, epoch).is_some() {
    return Err(ContractErrors::RewardEpochOutOfOrder);
```

```
}
```

In Soroban, instance-storage expiration archives the contract instance and makes the contract uncallable until it is restored. As a result, the shorter TTL on `LatestRewardEpoch` can cause the escrow contract to become unavailable even while reward-epoch entries are still retained in persistent storage, which creates a TTL mismatch.

Impact

If no relevant call refreshes the instance TTL in time, the escrow contract can become temporarily unavailable and all entry points will fail until the contract is restored. This introduces avoidable operational overhead and can delay normal reward-management flows even though the associated reward-epoch data may still be retained.

Recommendations

Extend the TTL of the `LatestRewardEpoch` to be the same as the reward-epoch TTL.

Remediation

This issue has been acknowledged by Sentora, and a fix was implemented in commit `eac8a35f` ↗.

3.9. Admin fund recovery lacks event emission

Target	escrow		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Both `withdraw` and `sweep_expired_epoch` are privileged functions that move assets from the escrow contract to the admin address. Unlike other security-relevant state changes in the same contract, these fund-recovery paths do not publish an event after a transfer.

```
// contracts/escrow/src/contract.rs#L85-L99
fn withdraw(e: Env, actions: Vec<(Address, u128)>) {
    let admin: Address = admin(&e, None).unwrap();
    admin.require_auth();
    for (asset, amount) in actions {
        // [...]
        if transfer_amount <= 0 {
            continue;
        }
        client.transfer(&e.current_contract_address(), &admin,
    &transfer_amount);
    }
    bump_instance(&e);
}
```

```
// contracts/escrow/src/contract.rs#L235-L265
fn sweep_expired_epoch(e: Env, asset: Address, vault: Address, epoch: u32,
    amount: u128) -> Result<(), ContractErrors> {
    let admin_addr = admin(&e, None).unwrap();
    admin_addr.require_auth();

    // [...]
    token_client.transfer(&e.current_contract_address(), &admin_addr,
    &amount_i128);
    reduce_committed_rewards(&e, &asset, amount);
    bump_instance(&e);

    Ok(())
}
```

The contract already emits events for admin rotation, reward-root publication, and reward claims. Missing events on admin withdrawals and expired epoch sweeps make these operationally important actions less visible to indexers, monitoring systems, and incident response tooling.

Impact

Off-chain systems must infer admin fund movements from token transfers alone and cannot reliably distinguish routine recovery from unexpected activity. This reduces auditability for privileged operations but does not by itself allow unauthorized fund movement.

Recommendations

Emit events from `withdraw` and `sweep_expired_epoch` that include the admin address, asset, amount transferred, and relevant context such as retained amount, vault, and epoch.

Remediation

This issue has been acknowledged by Sentora, and a fix was implemented in commit `bc747218` ↗.

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. Deployment verification

Zellic was asked to verify that the bytecode of the deployed contracts matches the bytecode compiled from the audited commit hash.

The deployment is based on commit `c29a8bf9` ↗. The following are the key changes made to the codebase between commit `a7955fcb` ↗ and the deployed commit `c29a8bf9` ↗:

- The remediations applied to the findings 3.4. ↗, 3.5. ↗, 3.7. ↗, 3.8. ↗, and 3.9. ↗.
- Contract metadata is added to both the escrow and vault contracts with the key `version` and the value `1.0.0`.

The contracts were deployed to the following addresses:

- escrow: CCZSNGVMEHRKMMMQGVDHAXT3KSWIMPEPRRTUE2GU3F6TW4XA2Y6EMHN3
- XLM vault: CA54LVHMAY7HGLMVPN4W72XJB4OGKVZBZX26FWN6JD4P3HJFWQUQEHJO
- USDC vault: CAHEWHOPPDBQYFMAOLDOXXGUX2BCR7EXP4CWYCRY3NEAJB35YPZMMJFF
- PYUSD vault: CAQRAXBU6G4AAX4BZ7R4WLB62TSVAQFS5ZXJDVXRLAU2NZ2ZTGU5QOYB

We have verified that the WASM hash of each deployed contract is consistent with the WASM hash generated from the contracts compiled from the audited commit. We verified only the on-chain storage state at deployment time. Subsequent operational configuration (e.g., reward settings) was not verified.

5. System Design

This section describes only the system-design changes introduced in commit 79139687 compared to commit a7955fcb.

5.1. Component: Escrow

Changed design elements

The escrow contract now exposes `update_admin()`. If an admin is already stored, the current admin must authorize the update. If no admin is stored, the function writes the new admin without authorization.

The escrow withdrawal path now converts the requested amount to `i128` with an explicit `WithdrawAmountTooLarge` failure path. It computes `balance - amount` and skips the transfer when the result is zero or negative.

Reward-root publication now tracks committed rewards per asset, and `set_rewards_root()` checks the token balance against already committed rewards plus the new epoch's `total_rewards`, then increments the committed total. Successful claims and expired-epoch sweeps reduce the committed total.

The escrow contract now exposes `sweep_expired_epoch()`, allowing the admin to recover an arbitrary amount for an epoch whose `RewardEpoch` storage entry is no longer present. The function requires the epoch to be no greater than the stored latest epoch for the vault, verifies the contract balance, transfers the requested amount to the admin, and reduces committed rewards for the supplied asset.

The Merkle hashing domain separators changed. Reward leaves now use the byte prefix `0x00`, and internal Merkle nodes are hashed with a leading `0x01` byte prefix before the two child hashes.

Changed storage

For `StorageKeys::CommittedRewards`, the type is Persistent. TTL extensions are extended by `add_committed_rewards()` and `reduce_committed_rewards()`, called in `set_rewards_root()`, `claim()`, and `sweep_expired_epoch()`. If expired, committed rewards for the asset become archived; claims, sweeps, and rewards-root publication will fail.

Test-coverage changes

The reviewed diff does not add new escrow tests for `update_admin()`, committed-reward accounting, the new Merkle domain separators, or `sweep_expired_epoch()`. It removes multiple preexisting escrow tests, including tests for epoch ordering, insufficient escrow balance, oversized reward amounts, Merkle leaf hashing, Merkle proof–root computation, and withdrawal edge cases.

Remediation added escrow tests for committed-reward bounded expired-epoch sweeps and admin recovery events emitted by `withdraw()` and `sweep_expired_epoch()`.

5.2. Component: Vault

Changed design elements

The vault constructor now accepts and stores an escrow contract address in instance storage. This adds a protocol-level relationship between the vault and escrow contracts, although the reviewed vault code does not yet use the stored escrow address during deposit or withdrawal execution.

The vault withdrawal path now checks the paused flag before allowing withdrawals. This changes the pause mechanism from deposit-only control to a control that can block both deposits and withdrawals.

Changed storage

For `CoreStorageKeys::Escrow`, the type is Instance. It reads through the `escrow()` storage helper. TTL extensions are extended by `bump_instance()`, called in `upgrade()`, `update_admin()`, `set_status()`, `deposit()`, and `withdraw()`. If expired, the entire instance storage becomes archived.

Test-coverage changes

The tests were updated to pass an escrow address into the vault constructor and to assert that the constructor stores it. No new tests were added for the changed withdrawal pause behavior, and several preexisting vault tests were removed in the reviewed diff.

6. Assessment Results

During our assessment on the scoped Pre-deposit Program files, we discovered nine findings. No critical issues were found. One finding was of high impact, three were of medium impact, one was of low impact, and the remaining 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.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

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.