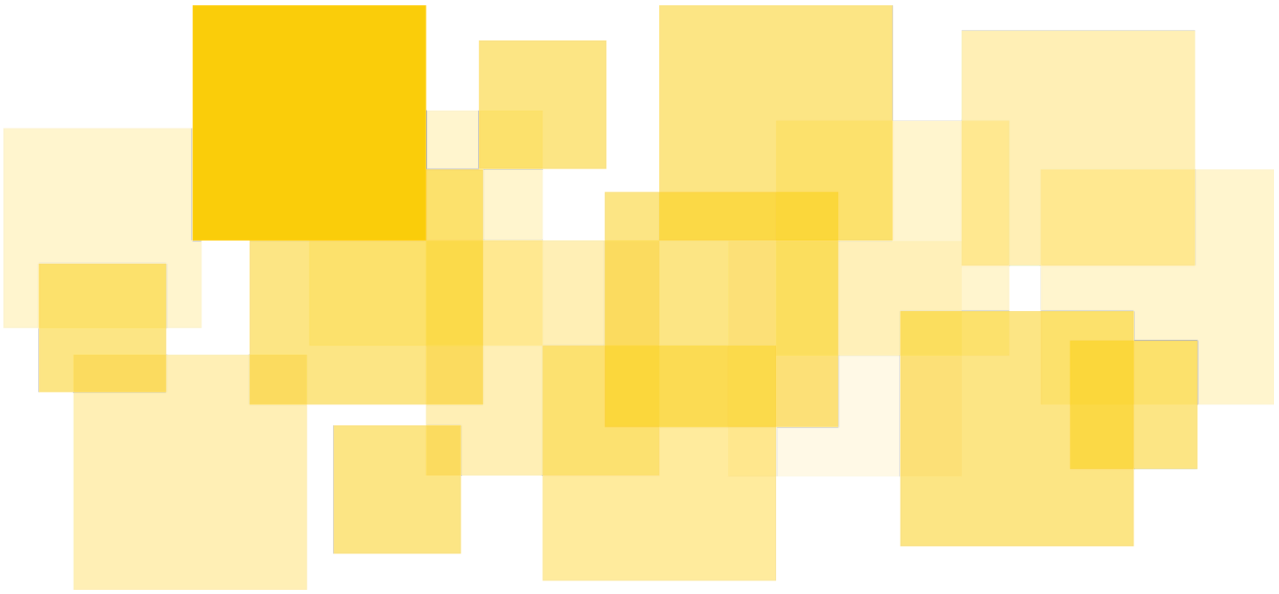
Security Audit Report

EquitX Stellar

Delivered: December 10, 2025



runtime verification

Table of Contents

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 that 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 carry 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 that impact a large quantity of funds.

Executive Summary

EquitX engaged Runtime Verification Inc. to perform a security audit of its smart contracts. The audit was conducted between October 13 and November 17, 2025. The objective was to assess the correctness and security of the EquitX on-chain implementation, identify vulnerabilities or design inconsistencies, and provide actionable recommendations to strengthen the protocol ahead of mainnet deployment.

EquitX enables the creation, management, and exchange of xAssets—synthetic on-chain representations of real-world equities. Users can deposit collateral, mint xAssets, trade them, manage collateralized debt positions (cDPs), participate in stability pools, and access real-time analytics directly from their wallet. The protocol aims to make traditionally inaccessible financial instruments globally available through a decentralized, transparent, and fully on-chain model.

The audit consisted of a manual code review of the smart contract codebase. No fuzzing, formal verification, or backend review was included in the scope of this engagement. Runtime Verification's analysis focused on contract logic, authorization flows, invariant correctness, token handling, collateralization mechanics, and other components critical to the safety of user positions and protocol funds.

Overall, code quality was strong, with clear structure, consistent patterns, and readable implementation. During the course of the engagement, the EquitX team responded promptly to all reported issues, and remediated them in a timely and effective manner. The fixes applied by the team generally demonstrated a solid understanding of the underlying risks and resulted in meaningful improvements to the protocol's safety.

The audit identified findings across a range of severities, from critical to informative. These findings are grouped into the following categories:

Findings – Issues with potential impact on protocol correctness, user funds, or core system behavior.

Informative Findings – General improvements, best-practice recommendations, and observations about clarity, maintainability, and future-proofing.

Runtime Verification performed targeted reviews of the implemented remediations and observed that the majority of fixes addressed the underlying issues as reported. As standard practice, we recommend that any substantial changes made after the audit period be reviewed as part of a follow-up assessment.

Overall, the EquitX team demonstrated strong responsiveness and a clear commitment to security. With the incorporated fixes, the audited codebase is materially more robust and better aligned with secure

development best practices.

Goals

The goal of the audit is threefold:

- Review the high-level business logic (protocol design) of the EquitX contracts based on the provided documentation and code;
- Perform a design review of the smart contract orchestrator and xAssets of the EquitX protocol;
- Review the low-level implementation of the individual Stellar-Soroban smart contract;
- Analyze the integration between abstractions of the modules interacting with the contract within the scope of the engagement and reason about possible exploitative corner cases;
- Perform best-effort reviews of the modifications consequent of this engagement's findings.

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 improve the safety and efficiency of the implementation.

Scope

The scope of this audit was initially limited to the code contained in the public GitHub repository provided by the EquitX team. The elements within scope are divided into three core components:

- equitx-project repository (Public Git Repository), commit `f0e45e7`, branch `main`:
  - `./contracts/xasset-orchestrator`: Contains the implementation of the deployer and manager of the xAsset contracts.
  - `./contracts/xasset`: Implements a SEP-0041 compliant token with the capabilities to manage a collateralized debt position.
  - `./contracts/data-feed`: Implements the test fixtures necessary for providing xAsset's contracts with token pricing information on tests.

The Data-Feed component was temporarily in scope for the initial weeks of the engagement, resulting in a limited number of findings. As the engagement progressed and findings were identified, the client was encouraged to perform a major refactoring of the Orchestrator and xAsset contracts, removing the Loam SDK from the contracts' code. In place of the Data-Feed contract, commits f79a130 and cb172a6 were placed in scope for a diligence review, ensuring functional equivalency between both versions of the contracts, with and without the Loam SDK.

The audit is strictly limited to the artifacts listed above. Frontend logic, deployment infrastructure, and third-party integrations are explicitly excluded from this engagement.

Commits addressing any findings presented in this report were also reviewed to verify that the identified issues were appropriately addressed prior to report finalization.

Methodology and Engagement Plan

Runtime Verification followed a structured methodology to evaluate the correctness and security of the EquitX smart contracts within the agreed engagement timeline. Although manual review cannot guarantee the discovery of all vulnerabilities, our approach was designed to maximize coverage and identify the most impactful risks.

Design Review and Invariant Identification

The engagement began with a high-level design review of the EquitX protocol, focusing on the lifecycle of xAssets, collateralization rules, liquidation mechanics, and trust assumptions.
To contextualize the intended behavior, we also referenced the Indigo Protocol whitepaper, which informed our understanding of expected invariants and economic properties relevant to synthetic asset systems.

This phase produced a set of functional and security invariants that guided the detailed review.

Manual Code Review

After establishing the protocol model, we performed a line-by-line review of the Soroban Rust codebase. This included examining:

- Correctness of collateral and xAsset accounting
- Authorization and permission boundaries
- Token handling and conversion logic
- Edge cases, boundary conditions, and potential state inconsistencies

Execution paths and state transitions were evaluated against the identified invariants to ensure alignment between the intended design and implemented behavior.

Tool-Assisted Analysis

To supplement manual review, we incorporated targeted automated analysis:

- **RV Audit Assistant** for AI-driven precondition analysis and vulnerability detection.
- **Almanax** for AI-assisted bug pattern detection and anomaly surfacing
- **Komet** for property-based fuzzing on selected components, primarily to validate proof-of-concept scenarios for certain identified issues

These tools enhanced coverage but were used in a supporting role, with manual reasoning driving the core assessment.

Platform Features And Logic Description

Orchestrator Contract

The Orchestrator is used to deploy multiple xAsset instances (xBTC, xETH, xUSDT, etc.) from a single WASM binary. The frontend and backend then use the Orchestrator's registry to resolve asset symbols to contract addresses, managing multiple xAsset contract instances. It exposes the following endpoints through the `IsOrchestratorTrait`:

- `init(xlm_sac, xlm_contract, xasset_wasm_hash)` - Initializes the Orchestrator with the XLM SAC address, XLM oracle contract address, and the WASM hash for xAsset contracts. This must be called once after deployment and administrator address setting.
- `update_xasset_wasm_hash(xasset_wasm_hash)` - Updates the stored WASM hash used for deploying new xAsset contracts. This allows upgrading the xAsset implementation without redeploying the Orchestrator.
- `deploy_asset_contract(asset_contract, pegged_asset, min_collat_ratio, name, symbol, decimals, annual_interest_rate)` - Deploys a new xAsset contract instance with the specified parameters. The deployment uses a deterministic salt based on the symbol to ensure unique addresses. The Orchestrator automatically initializes the deployed contract and registers it in the internal mapping of assets.
- `get_asset_contract(asset_symbol)` - Retrieves the contract address for a given asset symbol (e.g., "xBTC"). Returns an error if the asset doesn't exist.
- `set_asset_contract(asset_symbol, asset_contract)` - Manually registers an existing contract address to an asset symbol. Fails if the symbol is already registered.
- `set_existing_asset_contract(asset_symbol, asset_contract)` - Overwrites an existing asset symbol's contract address (dangerous operation). This should only be used when updating an existing symbol's contract.
- `upgrade_existing_asset_contract(asset_symbol)` - Upgrades an existing xAsset contract to the Orchestrator's current WASM hash. This allows upgrading deployed contracts without changing their addresses.

All mutating operations require admin authorization via `Contract::require_auth()`. The admin is set during deployment and can be set, updated, and fetched through the inherited Admin trait from Loam SDK.

xAsset Contract

The xAsset contract is a smart contract that combines a SEP-0041 fungible token with Collateralized Debt Position (CDP) management and Stability Pool mechanics. It enables users to mint synthetic assets (like xBTC, xUSD, xETH) by depositing XLM collateral, creating CDPs that must maintain a minimum collateralization ratio. The contract tracks each user's collateral, debt, interest accrual, and CDP status (Open, Insolvent, Frozen, or Closed).

The contract achieves its functionality through four trait implementations: `IsSep41` for standard token operations, `IsCollateralized` for CDP lifecycle management (opening, freezing, liquidating, repaying), `IsStabilityPool` for managing user deposits that absorb liquidated debt, and `IsCDPAdmin` for administrative configuration. It integrates with external price oracles via SEP-0040 to fetch real-time XLM and pegged asset prices, using these to calculate collateralization ratios and enforce minimum thresholds. When users open CDPs, the contract transfers XLM collateral from the user to itself via the XLM Stellar Asset Contract, mints the corresponding xAsset tokens to the user, and stores the CDP state.

The Stability Pool mechanism allows users to stake xAsset tokens to earn rewards from liquidations. When a CDP becomes undercollateralized and gets frozen, the liquidation process withdraws xAsset from the pool to repay debt. It distributes the CDP's XLM collateral proportionally to stakers based on their deposit amounts and tracking constants (`product_constant` and `compounded_constant`). The contract also implements an interest rate system where CDP owners accrue interest over time, payable in XLM rather than the principal token.

It achieves these functionalities by providing the following endpoints:

Opening & Querying CDPs:

- `open_cdp(lender, collateral, asset_lent)` - Creates a new CDP by depositing XLM collateral and minting xAsset tokens.
- `cdp(lender)` - Retrieves CDP information for a specific lender.

Collateral Management:

- `add_collateral(lender, amount)` - Increases collateralization ratio by depositing more XLM.
- `withdraw_collateral(lender, amount)` - Withdraws collateral if it doesn't bring CR below the minimum collateralization ratio.

Debt Management:

- `borrow_xasset(lender, amount)` - Mints additional xAsset against existing collateral.
- `repay_debt(lender, amount)` - Repays debt by burning xAsset tokens.

Interest Operations:

- `get_accrued_interest(lender)` - Returns detailed interest information, including amounts in both xAsset and XLM, plus approval amount for repayment.
- `pay_interest(lender, amount)` - Pays accrued interest in XLM (not principal).

Liquidation & Closure:

- `freeze_cdp(lender)` - Freezes undercollateralized CDPs (CR below the minimum collateralization ratio).
- `liquidate_cdp(lender)` - Liquidates frozen CDPs using Stability Pool funds, returning (collateral_liquidated, principal_repaid, new_status).
- `merge_cdps(lenders)` - Merges multiple frozen CDPs into one.
- `close_cdp(lender)` - Closes a CDP with zero debt and returns remaining collateral.

Deposit Management:

- `stake(from, amount)` - Creates an initial stake position in the pool (requires no existing position).
- `deposit(from, amount)` - Adds to an existing stake position.
- `unstake(staker)` - Withdraws the entire position, including rewards.

Position Queries:

- `get_available_assets(staker)` - Returns available xAsset and XLM rewards.
- `get_position(staker)` - Returns full staker position details.
- `get_constants()` - Returns current pool constants (product/compounded constants, epoch).
- `get_total_xasset()` - Returns the total xAsset deposited in the pool.
- `get_total_collateral()` - Returns the total XLM collateral in the pool.

Oracle & Price Feed Integration:

- `lastprice_xlm()` - Gets the latest XLM price from oracle.
- `lastprice_asset()` - Gets the latest pegged asset price.
- `decimals_xlm_feed()` / `decimals_asset_feed()` - Returns oracle decimal precision.

Admin Configuration Endpoints:

- `cdp_init(xlm_sac, xlm_contract, asset_contract, pegged_asset, min_collat_ratio, name, symbol, decimals, annual_interest_rate)` - One-time initialization with oracle addresses, MCR, token metadata, and interest rate.
- `set_xlm_sac(to)` / `set_xlm_contract(to)` / `set_asset_contract(to)` - Update oracle addresses.
- `set_min_collat_ratio(to)` - Adjust minimum collateralization ratio.

- `set_interest_rate(new_rate)` / `get_interest_rate()` - Manage annual interest rate in basis points.
- `get_total_interest_collected()` - Query total protocol interest revenue.
- `version()` - Returns contract version string.

Invariants

During the audit, invariants were defined and used to guide part of our search for possible issues with EquitX contracts. Based on prior specifications, client documentation, intended business logic, and collected references, we identified the following invariants:

Orchestrator Invariants:

- Only the EquitX administrator can modify Orchestrator state or deploy/upgrade contracts.
- Core properties of the orchestrator must remain invariant unless explicitly modified by designated functions.
- Each asset symbol managed by the Orchestrator can only be deployed once to prevent duplicate registrations.
- The assets map managed by the Orchestrator must accurately reflect all deployed contracts.
- Attempting to deploy an xAsset using the same symbol as the primary seed twice must produce the same contract address.
- All deployed xAsset contracts must be owned by the Orchestrator's admin.
- After deployment, the xAsset contract must be successfully initialized with all required parameters.
- All asset upgrades must use the Orchestrator's current WASM hash.

xAsset invariants:

- Every Open CDP must maintain `collateralization_ratio` >= `min_collat_ratio` at all times.
- A CDP with a collateralization ratio smaller than the minimum allowed collateral ratio must have the status Insolvent or Frozen, (never Open).
- Each address can have at most one CDP per xAsset contract.
- `sum(all balances) = sum(all CDP asset_lent) + total_xasset (in stability pool)`.
- All token balances must be >= 0.
- `total_xasset >= 0` and equals the sum of all staker deposits in the current epoch.
- `total_collateral >= 0` and represents accumulated XLM from liquidations.
- `product_constant` must decrease or stay constant during liquidations (never increase).
- `compounded_constant` must increase or stay constant (never decrease).
- Staker positions with `epoch != current_epoch` have zero available deposits.
- Accrued interest must be non-negative and calculated as: `principal * rate * time_elapsed / (SECONDS_PER_YEAR * BASIS_POINTS)`.
- Interest must be paid before principal can be repaid.
- Interest payments reduce `accrued_interest.amount` and increase `accrued_interest.paid`, and the payment amount cannot exceed the current accrued interest.

- Interest must be fully paid before principal liquidation begins.
- Only the EquitX administrator can modify the xAsset's core state.
- Transfers using allowances cannot exceed the approved amount.
- All collateral rate calculations must use freshly fetched oracle prices.
- Collateral rate calculations must adequately account for decimal differences between XLM and xAsset feeds.

We highlight that these invariants were raised during a high-level design review of the protocol, and used as initial security guidelines for identifying potential issues with the reviewed contracts. They were also created assuming that Stellar's native features used by the protocol are sound regarding safety and security.

Furthermore, although these invariants guide our analysis and research, any other behavior diverging from the expected protocol business logic was noted and brought to the client's attention.

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] xAsset Token Transfers Can Exceed The Sender Balance

Severity: High | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The xAsset contract, which combines a SEP 41-compliant fungible token with a Collateralized Debt Position management system, contains a fundamental flaw in its token transfer logic. The contract provides the `transfer_from` function, allowing a delegated entity to transfer xAsset tokens. However, the current implementation of this function only validates the delegated allowance amount and ensures that the transfer value is positive. It critically fails to verify whether the source account has a sufficient balance of xAsset tokens to complete the operation, resulting in a severe accounting error.

This oversight is exacerbated by the development choice to utilize `i128` signed integers for balance tracking. Since `i128` allows for both positive and negative values, the absence of a balance enforcement mechanism means that a user can successfully transfer an amount of xAsset greater than their current holdings. This transaction causes the source account's token balance to become negative.

The ability to create an artificial negative token balance establishes a direct path for exploiting the protocol's primary functionalities related to the collateralization of assets, manipulating user positions, and potentially causing severe financial negative impacts to xAsset token holders.

Recommendation

To solve this critical security issue, implement a strict balance check before authorizing any token transfer. This enforcement must ensure that the source account's token balance is greater than or equal to the requested transfer amount.

Status

This finding has been addressed in commit ID `091d3aa` on the project's main branch. This issue has been resolved by following the recommendation above.

[A2] It Is Possible To Front-Run The Initialization Of The Orchestrator Contract

Severity: Medium | Difficulty: High | Recommended Action: Fix Design | Addressed by client

Description

The current deployment and setup process for the orchestrator smart contract is vulnerable to front-running because it relies on a three-transaction sequence for complete initialization.. This sequence involves: 1) deploying the contract, 2) setting the contract administrator (via the inherited `Admin` trait), and 3) calling the `init` function with core parameters. The time interval between deployment and the intended administrator setting transaction creates a race condition exploitable by an adversary.

An attacker can monitor the protocol administrator's operations, anticipating attempts to deploy and initialize instances of the orchestrator. Upon observing the deployment attempt, they can immediately craft and broadcast a transaction to call the administrator setting function, offering a higher fee to prioritize its inclusion in a block ahead of the intended transaction. This allows the attacker to preemptively register their address as the contract administrator. Once unauthorized control is established, the attacker can then initialize the orchestrator with potentially malicious parameters, thereby compromising the security of the entire xAsset deployment ecosystem.

Recommendation

To secure the orchestrator and mitigate this risk, the contract logic must be modified to implement a constructor pattern, a recent introduction to Soroban. This pattern mandates that all critical setup actions, including the definition of the administrator and the configuration of essential contract parameters (XLM SAC address, XLM contract address, xAsset WASM hash), be performed in a single, atomic function call upon deployment. Consolidating the setup prevents external actors from interjecting and taking control, ensuring that the contract is initialized securely by the intended party in a single, atomic transaction.

Status

This finding has been addressed in commit ID `f79a130` on the project's main branch. The issue has been resolved by migrating from the deploy-initialize pattern to using constructors, as recommended.

[A3] It Is Possible To Override Properties Of The Orchestrator Supposed To Be Immutable

Severity: Medium | Difficulty: High | Recommended Action: Fix Design | Addressed by client

Description

The Orchestrator contract, central to managing and deploying all xAsset token instances, is configured via the `init` function that accepts core protocol addresses and the xAsset WASM hash. A security vulnerability exists because this initialization function lacks a control mechanism to prevent its execution after the first successful call. The implementation allows the administrator to invoke `init` indefinitely without any restriction or check on the contract's current state.

This lack of execution control means that values intended to be set only once, such as the crucial xAsset WASM hash or the external contract addresses, can be arbitrarily overwritten. Any subsequent call to `init` will overwrite the existing configuration, thus violating the principle of parameter immutability. An administrator (whether compromised or acting maliciously) could use this flaw to change the WASM hash, potentially directing future deployments to an incorrect or malicious xAsset contract version. This compromises the security and stability of the entire system.

Recommendation

To mitigate this severe issue, the Orchestrator contract must be modified to ensure the `init` function executes only once throughout the contract's lifecycle. Alternatively, the constructor pattern, a recent introduction to Soroban, should be implemented.

Status

This finding has been addressed in commit ID `f79a130` on the project's main branch. The issue has been resolved by migrating from the deploy-initialize pattern to using constructors.

[A4] Malicious Users Can Grow Their Balances By Transferring To Themselves

Severity: High | Difficulty: Low | Recommended Action: Fix Design | Addressed by client

The xAsset contract exhibits a critical flaw in its underlying token accounting mechanism when a user performs a self-transfer. Specifically, the contract's transfer logic fails to validate whether the source and destination accounts are identical. This omission permits a self-transfer, an action that should result in a zero net balance change but instead triggers an erroneous inflation of the account's holdings.

This vulnerability arises from the non-atomic, sequential balance modification logic within the contract's `transfer_internal` function. The function first executes a deduction from the sender's balance and subsequently executes an addition to the receiver's balance. Notice that no update in the stored balance is actually performed in between these two operations, and yet, to perform these calculations, the balances are fetched from storage. The storage update only happens after the calculations. When the sender and receiver are at the same address, the balance is first correctly reduced by the amount transferred, but then incorrectly increased by the amount transferred. This results in the account's final balance being inflated by the transferred amount, effectively creating uncollateralized and unbacked xAsset tokens, enabling unlimited token minting.

To illustrate this, consider that Alice holds 1,000 xAsset tokens. If Alice transfers all its balance to itself, its balance will be calculated and updated twice. Once to zero as the sender (1,000 - 1,000), and once to 2,000 as the destination (1,000 + 1,000).

Recommendation

To eliminate this token duplication vulnerability, the xAsset contract must be patched to introduce a verification step in its transfer functions, ensuring that the source and destination of transfers must be different. Additionally, the storage instance of a user's balance must be updated immediately after the calculation is complete. No other operation must be performed between the user's balance calculation and storage modification.

Status

This finding has been addressed in commit ID `285ebfe` on the project's main branch following the above recommendation.

[A5] Delegated Operations Need Signatures From Both Spender And From Accounts

Severity: Low | Difficulty: Medium | Recommended Action: Fix Design | Addressed by client

Description

The xAsset contract implements delegated spending functions, including `transfer from` and `burn from`, which are essential for enabling third-party interaction with a user's funds via the allowance mechanism. A fundamental flaw exists in the authorization flow for these delegated operations. While the functions correctly require the signature of the `spender`, they also incorrectly necessitate the signature of the `from` account (the original token owner). This requirement defeats the core purpose of delegated spending, which is to allow a single authorized party to execute transactions on behalf of the owner without further direct intervention.

This issue stems from the internal logic of the delegated functions. The `transfer_from` and `burn_from` functions, which require the `spender` signature, call the `decrease_allowance` function, which requires a signature from the `from` account owner.

Recommendation

To rectify this issue, the contract must be refactored to separate internal logic from external authorization. The protocol team should implement a new internal helper function, for instance, `decrease_allowance_internal`, to perform the allowance modification without enforcing the `from` account's signature. This function would then be used by `transfer_from`, `burn_from`, and `decrease_allowance`.

Status

This finding has been addressed in commit ID `4591680` on the project's main branch following the above recommendation.

[A6] Overflow And Underflow Possibilities When Handling Token Balances

Severity: Low | Difficulty: High | Recommended Action: Fix Code | Addressed by client

Description

The xAsset contract relies on `i128` signed integers for tracking and manipulating user balances across various operations, including deposits and transfers. A critical vulnerability exists because the contract fails to implement sufficient overflow protection when adding new amounts to existing balances (e.g., in a transfer from A to B: `B.balance + amount`). If a user's current balance is near the maximum positive limit of the `i128` data type, adding even a small, valid amount could cause the balance to exceed the maximum value, resulting in an arithmetic overflow.

While token transfers illustrate this issue, this behavior is a risk in any operation manipulating `i128` state variables.

Recommendation

To mitigate this issue, implement checked arithmetic operations, which are natively available within the development framework. These checked operations will automatically detect and gracefully revert the transaction if the result of an addition operation exceeds the maximum positive value of the `i128` integer.

Status

This finding has been addressed in commit ID `3f321c1` on the project's main branch.

[A7] It Is Possible To Have Negative Allowances

Severity: Low | Difficulty: Medium | Recommended Action: Fix Code | Addressed by client

The xAsset contract's token allowance mechanism contains a flaw in the `decrease allowance` function that undermines the security of delegated spending. This vulnerability permits the allowance, which is tracked as a signed integer, to be manipulated into a negative value (an integer underflow). Specifically, when the amount requested to decrease the allowance exceeds the existing authorized allowance, the subtraction operation results in a negative remainder. For example, reducing an allowance of 10 tokens by 20 tokens results in an invalid allowance of -10 tokens. This creation of a fraudulent negative limit compromises controls, allowing a delegated spender to proceed with subsequent transfers under a completely corrupted authorization status.

This flaw is particularly dangerous because the integrity of the token standard's delegation system relies on allowances being a non-negative measure of permissible spending. A negative allowance is an invalid state that can lead to unexpected and undefined behavior in downstream functions that check or rely on the allowance value.

Imagine that Alice, who previously provided Bob with an allowance, wants to decrease the allowance as much as possible, now being sure how much of it Bob has already used. Alice then provides an intentionally large number as a parameter to `decrease_allowance` to be sure Bob wouldn't have an allowance anymore, causing his allowance to become negative. Suppose Alice wants to make Bob a delegated entity capable of transferring 10 tokens. In that case, she must increase his allowance by the current negative allowance amount plus 10 tokens, creating a new operational overhead to Alice.

Recommendation

To resolve this issue, the `decrease allowance` function must be secured with a check to prevent integer underflow. If attempting to decrease an allowance below zero, the new allowance for the delegated entity should be set to zero instead.

Status

This finding has been addressed in commit ID `45ce2bd` on the project's main branch. This issue has been resolved by enforcing that the new allowance should be zero or greater, leading to [A10] A User's Allowance Can Never Be Fully Used.

[A8] The Admin Can Arbitrarily Mint Tokens Supposed To Be Minted When Managing CDPs

Severity: High | Difficulty: High | Recommended Action: Fix Design | Addressed by client

The xAsset contract, which is designed as a Collateralized Debt Position (CDP) system, includes a `mint` function secured only by an administrator authorization check. This implementation allows the contract administrator to call the function and create new xAsset tokens without requiring corresponding XLM collateral deposits. This is a severe design flaw that bypasses the core economic stability mechanism of the CDP system, which mandates that all synthetic assets must be backed by locked collateral.

This capability poses an existential threat to the protocol's solvency and trust model. If the administrator is compromised or acts maliciously, they can instantly `mint` an arbitrary supply of xAsset tokens. This results in immediate token inflation, which dilutes the value of all existing xAsset tokens held by users and the Stability Pool. Critically, this also unbalances the protocol's collateralization ratio.

Finally, even with a guarantee that the administrator cannot be compromised, this capability still creates a significant trust issue for potential protocol users.

Recommendation

To resolve this issue, the independent `mint` function accessible to the administrator must be eliminated or reimplemented to prevent its default usage.

Status

This finding has been addressed in commit ID `f79a130` on the project's main branch. This issue has been resolved by removing the default `mint` implementation from the xAsset contract.

[A9] Function Overrides Wrong Field

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The refactored version of the xAssets contract, which does not use the Loam SDK, contains an equivalency issue in the `set_xlm_contract` function.

In its new version, it incorrectly assigns the new address to `state.xlm_sac` instead of `state.xlm_contract`. This is a clear implementation bug: the function is supposed to set the XLM oracle price feed contract, but instead modifies the XLM Stellar Asset Contract (SAC) address.

Recommendation

Correct the implementation of `set_xlm_contract`, to update `state.xlm_contract` instead of `state.xlm_sac`.

Status

This finding has been addressed in commit ID `5c767af` on the project's main branch by following the above recommendation.

[A10] A User's Allowance Can Never Be Fully Used

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The `decrease_allowance`, `transfer_from`, and `burn_from` functions share an issue when attempting to reduce a delegated entity's allowance to zero.

These three functions use `decrease_allowance` to reduce the delegated entity's allowance, which uses `set_and_extend_allowance` to modify the storage instance for the allowance field. This last function uses `assert_positive` on the allowance amount. This causes transactions to revert if the allowance is depleted entirely, as `assert_positive` enforces that the value must be strictly greater than zero.

Let's say that Alice gives Bob an allowance of 1,000 tokens, and Bob uses `transfer_from` to send all 1,000 tokens from Alice to another address. This transfer attempt will fail as `set_and_extend_allowance` will be called with an allowance of 0, and `assert_positive`, which enforces that the received value must be greater than 0, will cause an error to be triggered.

Recommendation

Instead of using `assert_positive` when handling the allowance amount, the contract should enforce that this amount is non-negative.

Status

This finding has been addressed in commit ID `421e263` on the project's main branch by following the above recommendation.

[A11] Missing Logic for Utilizing the Protocol Revenue From Interest and Fee Payments

Severity: High | Recommended Action: Fix Design | Not addressed by client

Description

The contract collects XLM through interest and fee payments, but there is currently no logic to route, distribute, or otherwise make use of these funds. As a result, all accumulated XLM remains permanently locked inside the contract. Since this revenue is part of the protocol's economic design, the contract must explicitly define how these funds are used, whether by distributing it to the stability pool or allocating it to stakeholders. Without such logic, the protocol's revenue flow remains incomplete and economically ineffective.

Status

The client acknowledges the issue and plans to implement a mechanism to handle the accumulated XLM balance in a future protocol upgrade. The intention is for the balance to be managed by a DAO, ensuring proper routing and use of protocol revenue. No detailed implementation timeline has been provided yet.

[A12] Incorrect Handling of Interest During Liquidation

Severity: Medium | Recommended Action: Fix Design | Addressed by client

Description

A CDP carries two kinds of debt: principal debt, and interest debt. During liquidation, all the debt a CDP has should be settled. Currently, the `liquidate` function performs calculations related to the interest debt, but these calculations are never reflected in any token transfers or burn operations. Additionally, the `accrued_interest` field in the CDP struct is updated as if the interest were actually paid, which results in incorrect information being stored. As a result, the interest debt is not actually paid out, and the protocol does not receive the corresponding revenue.

Recommendation

During liquidation, the accrued interest amount should be converted to XLM and paid out from the CDP's collateral. This portion of the collateral should be retained by the protocol as revenue. The remaining collateral should then be distributed to the stability pool, and the principal debt should be covered by burning xAsset from the pool.

The risk is that a large interest amount could consume too much collateral, and the collateral left for the stability pool might end up being worth less than the xAsset principal burned. To avoid harming stakers, the protocol should impose a safety cap on how much collateral can be used for interest payments: only the collateral value that exceeds the principal debt (in XLM terms) may be consumed. This guarantees that the portion of collateral needed to cover the principal is never touched, and ensures the pool always receives XLM value at least equal to the value of the xAsset it gives up.

Status

This finding has been addressed in commit ID `a8ef097` on the project's main branch by following the above recommendation.

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

Severity: Informative | Recommended Action: Fix Code | Addressed by client

The following notes present suggestions to improve the protocol's code quality, clarity, and business logic alignment with best practices. These do not constitute vulnerabilities but are advised recommendations for awareness or implementation.

1. (Orchestrator) In the Main contract, the core authorization validation should be moved from Line 118 to Line 117, making it the function's first operation. It is a best practice to perform all validations related to authorization and authentication before fetching state values or executing core logic.
2. (Orchestrator) The `expect()` in line 209 will never be triggered. It can be replaced by an unwrap. This is coming from a code conciseness and reduced binary size/assembly size perspective.
3. (xAsset) There is more than one section of implementation for the `Token` struct. Ideally, they should be grouped together.
4. (xAsset) Significant events related to the management of CDPs, such as minting and burning tokens, should emit events to facilitate off-chain user management and monitoring.
5. (xAsset) The `matches!` macro is used for CDP status comparisons. Consider using the equality operator (e.g., `cdp.status == CDPStatus::Closed` instead of `matches!(cdp.status, CDPStatus::Closed)`) for improved readability, given that `CDPStatus` implements the `PartialEq` trait.

Status

These findings have been addressed in the following commit IDs on the project's main branch:

1. f79a130
2. f79a130
3. cb172a6
4. 694399b
5. 96469cf

[B2] Error Handling When Attempting To Transfer Assets Can Be Misleading

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

The xAsset contract exhibits flawed error handling within its interest payment functions, specifically `pay_interest_from` and `pay_interest`. When attempting a required external XLM Stellar Asset Contract transfer, the code incorrectly maps an unlikely `ConversionError`, which occurs if the successful call result cannot be interpreted, to the business logic error `InsufficientApprovedXLMForInterestRepayment`.

This failure to accurately distinguish between a low-level data conversion problem and a high-level approval deficiency may mislead users and developers. Since a conversion error from the standard XLM SAC is improbable, this erroneous mapping prevents the contract from correctly identifying and reporting genuine underlying transfer failures, such as a true lack of funds or insufficient allowance, impacting transaction diagnosis.

Recommendation

To ensure correct and transparent reporting, the contract must be updated to precisely analyze the error result returned by the external XLM SAC transfer call. The recommended fix is to ensure the contract analyzes the specific error within the transaction result's `Err(_)` case, correctly mapping it to the appropriate application-level error, and handles the rare `ConversionError` separately.

Status

This finding has been addressed in commit ID `96469cf` on the project's main branch by following the above recommendation.

[B3] An Attempt To Pay Zero Interest Will Lead To Full A Interest Payment

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

The xAsset contract's `apply_interest_payment` function suffers from a design flaw where the numeric value of zero is overloaded to function as a special command. Specifically, setting the parameter representing the amount of xAssets in this function to zero is incorrectly interpreted as an instruction to "pay all accrued interest," rather than a simple zero-value payment attempt. This practice of assigning a special meaning to a standard quantity introduces unnecessary logical complexity, making the code less intuitive and increasing the risk of error by developers and users. This conflation of a quantity with a command can lead to the accidental execution of the "pay all" logic by users intending to make no payment, thus compromising clarity and increasing the risk of misuse throughout the protocol.

Recommendation

To resolve this design flaw and enhance both safety and clarity, the function signature must be updated to clearly separate payment amounts from special commands. The protocol team should change the numeric amount in xAsset amount parameter to an optional type, `Option<i128>`, ensuring that a `None` value explicitly communicates the instruction to pay all accrued interest, thus eliminating the dangerous overloading of the zero value.

Status

This finding has been addressed in commit ID `98b0998` on the project's main branch. This issue has been resolved by removing the check on the interest amount, and an `Option<i128>` parameter is now used to explicitly indicate whether the full interest payment should be executed.

[B4] Data-Feed Issues

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

Although within the scope of the engagement, as detailed in the Scope section, two issues have been identified with the Data-Feed test fixtures.

1. (Data Feed) The `sep40_init` function implemented for SEP-0040 permits the contract to be initialized multiple times, potentially overriding core values that should be immutable.
2. (Data Feed) `IsSep40Admin` trait contains comments specifying security properties of its operation. One states that assets within that contract cannot be overridden, or added twice. However, no validations enforce this behavior in the `add_assets` function (present in the Data-Feed contract), thereby bypassing the contract's specified security properties.

Status

1. This finding has been addressed in commit ID `b61f85f` on the project's main branch. The issue has been resolved by shifting from the deploy-initialize pattern to using constructors.
2. This finding has been addressed in commit ID `892ce2e` on the project's main branch. This issue has been resolved by adding a validation step to `add_assets` to prevent duplicate entries.