



Security Audit Report

Trustless Work Stellar

Delivered: November 7, 2025

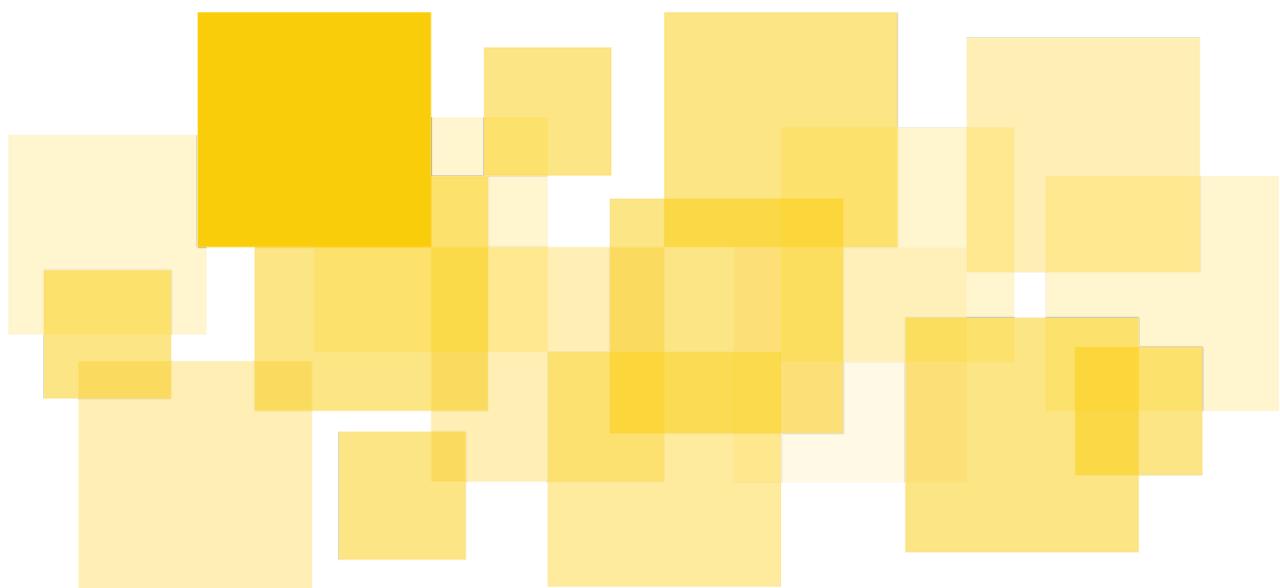


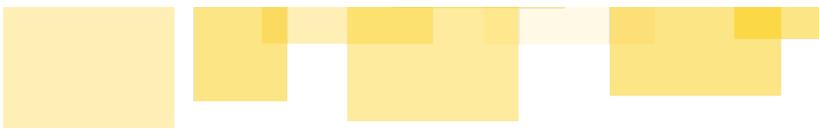
Table of Contents

- Disclaimer
- Executive Summary
- Goals
- Scope
- Methodology and Engagement Plan
- Platform Features and Logic Description
 - Smart Escrow
 - Smart Escrow Lifetime and Operations
 - Multi-Release Version Extension
 - Trustless Work's Backend
 - Core Architecture Components
 - Backend Service Architecture
 - API Interface
 - Data Management and Persistence
 - Transaction Flow Management
 - Authentication and Security
 - External Dependencies
 - Client Integration
- Invariants
 - General
 - Roles
 - Status state changes
 - Funds
- Findings
 - [A01] Dispute resolver and platform can call with arbitrary `trustless_work_address` and redirect TW fees to another address
 - [A02] Maximum relative amount of fees not checked at initialization time
 - [A03] Initialization possible with pre-approved milestones
 - [A04] Bypass of the Escrow Initialization Validations
 - [A05] Front-running attack by a malicious `platform` on `fund_escrow` with `update_escrow`

- [A06] Smart Escrow uses signed integer type `i128` without sign checks
- [A07] The Approver Can Disapprove Milestones
- [A08] Exceeding Assets Will Be Permanently Locked in The Multi-Release Smart Escrows
- [A09] Milestones can both be released and dispute-resolved in the multi-release smart escrows
- Informative Findings
 - [B01] Gas optimizations
 - `token_client.balance(&contract_address)` called twice in `core/dispute.rs:46`
 - Redundant out-of-bounds check of milestone index
 - Empty milestones checks outside of Escrow initialization and update
 - Unused admin
 - Unnecessary use of the function `clone`
 - Conditional at `core/escrow.rs:17-21`
 - Remove `signer` used in `core/escrow.rs:102`
 - Disallow `resolve_dispute` with `total_funds = 0`
 - Unnecessary checks in `resolve_dispute`
 - Iterative creation of Soroban vectors instead of patching in-place
 - Unnecessary write to storage
 - Unused enumerator iterator and vector length check in for loop
 - [B02] Initialization possible with zero milestones
 - [B03] At initialization, the `decimals` attribute of `Trustline` is not validated
 - [B04] `validate_release_conditions` returns wrong error when checking whether funds were already released
 - [B05] `validate_release_conditions` does not check whether the Smart Escrow has already been resolved
 - [B06] `dispute_resolver` should consent to Smart Escrow prior to funding the escrow
 - [B07] Smart Escrows Are Deployers Despite Having Constructors
 - [B08] Disputed Funds Don't Go Back to Funders
 - [B09] The Trustless Work Platform Does Not Handle Storage Archival
 - [B10] The Milestone Updates Phase Can Be Skipped
 - [B11] It Is Possible To Change The Platform Address Of A Smart Esrow
 - [B12] Differences Between the Single and Multi-Release Escrows
- Backend Findings

- 
- [TS01] In-Memory Queue Storage Causing Data Loss and Scaling Issues
 - [TS02] Open CORS Policy Allowing Unauthorized Cross-Origin Requests
 - [TS03] Missing Authentication Guards on Critical Endpoints
 - [TS04] Authorization Bypass in Escrow Repository
 - [TS05] Information Disclosure Through Error Messages
 - [TS06] HTTP Connections Allowed in Production Environment
 - [TS07] Unsafe HTTP Methods for State-Changing Operations
 - [TS08] Type Safety Issues - From 'any' Types to DocumentData Casting
 - [TS09] Incorrect Validation Decorators for Numeric Fields
 - [TS10] Client-Side Timestamps Creating Data Inconsistency
 - [TS11] Missing Input Validation and Type Annotations
 - [TS12] Unreliable Type Guard Functions and Interface Design Issues
 - [TS13] Environment Configuration Management Issues
 - [TS14] Code Quality and Documentation Issues
 - [TS15] Data Architecture and Repository Pattern Issues
 - [TS16] Development Tooling and Code Quality Setup
 - Install ESLint and related packages
 - Install Husky for git hooks
 - Install Prettier and integration
 - Install lint-staged for efficient pre-commit processing
 - [TS17] Backend allows for submitting any pre-signed transaction to the Stellar blockchain
 - [TS18] Singleton State Sharing in Transaction Builder
 - [TS19] Insecure private key storage in singleton service instance variables
 - [TS20] Inefficient database querying
 - [TS21] Notifications are created every hour once they start getting generated
 - [TS22] No notifications are being created for multi-release escrows
 - [TS23] Use of floating-point `number` type for critical values
 - [TS24] API key does not meet security standards
 - [TS25] Not using synced blockchain state in database
 - [TS26] API provided values are silently overridden with default values
 - [TS27] Login authentication procedure does not authenticate user
 - [TS28] The backend relies on data in `pendingWriteQueue` and `getTransaction` instead of events and blockchain storage

- [TS29] Missing validation whether user-submitted transactions were actually submitted to and processed by the blockchain
- [TS30] Recommendation: use self-hosted Stellar nodes as the Stellar API endpoint
- [TS31] `set-trustline` endpoint uses a private key as an argument
- [TS32] Issued JWT tokens never expire
- [TS33] Potentially never-ending loop querying the Stellar API
- Limitations and Recommendations for Follow-Up Audit



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

Trustless Work engaged [Runtime Verification Inc.](#) to perform a security audit of its smart contracts. The audit was conducted between August 5 and September 12, 2025. The objective was to assess the implementation's security and correctness, identify exploitable vulnerabilities, and provide recommendations to enhance the system's reliability.

The Trustless Work project enables trustless payments via smart contracts, also referred to as Smart Escrows, securing funds in escrow until clients approve milestones. Stablecoins like USDC ensure stability and ease of use, but the protocol has no limitations regarding what tokens can be used as the escrow's trustline.

The escrows work as vaults, storing assets and funds in a smart contract. The release of these assets depends on the logic of the smart contracts themselves as well as the actions of users with designated roles, representing authorities with specific responsibilities in the Trustless Work protocol.

The audit process involved a comprehensive smart contract codebase review, focusing on manual inspection and formal verification techniques. Runtime Verification utilized several techniques related to formal specification generation, invariant analysis, and testing to test the system's behavior under various conditions rigorously. This included the development and analysis of key invariants to ensure the system's integrity across all state transitions.

In addition, the Trustless Work team provided a TypeScript backend codebase for review. This component was not subject to a full audit; instead, Runtime Verification performed a three-week, time-boxed design review focused on major architectural and security concerns. Within this limited scope, we identified a significant number of issues, many of them recurring in pattern and continuing to surface throughout the engagement. This suggests that additional vulnerabilities or edge cases may exist beyond those captured in this report.

- Accordingly, we strongly recommend that the Trustless Work team:
- Address all recommendations from this report,
- Perform an additional internal security review focused on similar patterns and classes of vulnerabilities, and
- Consider allocating further resources to an extended or follow-up audit of the backend.

It is important to note that while the smart contracts were fully audited, the backend review was limited in scope and duration. As such, Runtime Verification cannot provide the same level of



assurance for the backend component as for the audited smart contracts.

During the course of the audit, a significant number of issues were identified and subsequently addressed by the Trustless Work team. While many of these fixes appear to resolve the issues as reported, several introduced substantial refactoring or architectural changes that could not be fully reviewed within the audit timeframe. Runtime Verification performed targeted spot-checks of the remediations, but did not conduct a full re-audit of all modified components. As a result, we recommend that the updated codebase, including all remediations, undergo a comprehensive follow-up audit before the protocol secures significant value.

The audit resulted in findings ranging in severity from critical to informative. These findings have been organized into the following sections:

- **Findings**: considers findings that constitute threats to the on-chain protocol's health or user/protocol funds;
- **Informative Findings**: considers findings that constitute general improvements for the on-chain protocol or potential enhancements to its overall design and implementation;
- **Backend Findings**: considers findings discovered during the design review of the Trustless Work protocol's off-chain backend.



Goals

The goal of the audit is threefold:

- Review the high-level business logic (protocol design) of the Trustless Work contracts and backend based on the provided documentation and code;
- Review the low-level implementation of the individual Stellar-Soroban smart contract;
- Perform a design review of the smart contract orchestrator of the Trustless Work system, represented by the TypeScript code in scope;
- 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.

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 is limited to the code contained in public and private GitHub repositories provided by the Trustless Work team.

- Escrow contracts ([Public Git Repository](#))
 - Commit: `5d4669d69ecdf1a8c788b5e644078f797f818850` , branch `develop` ;
 - `./contracts/` : Contains the implementation of the core on-chain logic of the Trustless Work protocol.
- Escrow contracts, Multi Release Version ([Public Git Repository](#))
 - Commit: `68d9a9920a50a60223ec85c20400ee8af72d6e4e` , branch `multi-release-develop` ;
 - `./contracts/` : Contains the implementation of the core on-chain logic of the Trustless Work protocol, with additional support for milestone-based payment releases.
- Orchestration Scripts/Backend (Provided in a private repository)
 - Commit: `8087379c690324185aaef24e14a4d1f39aef840b` , branch `develop` ;
 - `./src/` : Contains the TypeScript code implementing an orchestration system that interfaces users with the on-chain elements of the Trustless Work platform.

The codebase under review consists of 944 lines of Rust code written for the Stellar-Soroban ecosystem and 10,007 lines of TypeScript code, consisting of the backend orchestration scripts of the Trustless Work platform. In preparing for the audit, Runtime Verification referenced comments provided in the code, publicly available documentation, and supplemental materials shared by the Trustless Work team.

A low-level audit of the protocol's on-chain components and a design review of the TypeScript elements have been performed, focusing on validating core security elements identified during the analysis.

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

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



Methodology and Engagement Plan

Although manual code review cannot guarantee the discovery of all possible security vulnerabilities, as noted in our Disclaimer, we followed a structured and thorough approach to maximize the effectiveness of this audit engagement within the agreed timeframe.

The audit spanned five calendar weeks and three days, with each phase of the process designed to identify and validate both high-level and low-level security concerns.

During the first week, we conducted a high-level design review of the Trustless Work's Smart Escrow. We analyzed the architecture, key trust assumptions, and the security implications of interacting with various liquidity venues on the Stellar network. Particular emphasis was placed on identifying core invariants that should be upheld by the protocol under all valid inputs and runtime conditions.

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

- Modeled sequences of logical operations, considering the limitations enforced by the identified invariants, checking if all desired properties hold for any possible input value;
- Manually built high-level function call maps, aiding the comprehension of the code and organization of the protocol's verification process;
- We used static analyzers and AI-assisted security analyzers such as [Scout](#), [cargo audit](#), and [Almanax](#) to identify commonly identifiable issues;
- Created abstractions of elements outside the scope of this audit to build a complete picture of the protocol's business logic in action.

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

For the orchestration scripts implemented in TypeScript, we timeboxed a three-week effort to understand the design of the system, comprehend its interaction surface, generate abstractions of it, and rigorously validate them against possible threats imposed by malicious users, which constituted a design review of the protocol's backend system.

Finally, we conducted rounds of internal discussions with security experts over the codebase and platform design, aiming to verify possible exploitation vectors and identify improvements for the



analyzed contracts. Code optimizations have also been discovered as an outcome of these research sessions and discussions.

Findings in this report stem from a combination of:

- Manual inspection of the source code.
- Design-level reasoning about protocol behavior and system invariants.
- Analysis of abstractions and threat models against the generated invariants.

Throughout the audit engagement, vulnerabilities and edge cases were reported to the Trustless Work team in real-time, allowing for iterative discussion and clarification, which accelerated the fix-and-verify cycle. At the end of the engagement, an additional version of the smart escrow contract was investigated, following the strategies previously used to analyze the initial contract version.

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

Platform Features and Logic Description

The Trustless Work system is a decentralized payment infrastructure built on the Stellar blockchain that enables secure, milestone-based transactions between clients and service providers. This system eliminates the need for traditional intermediaries by using smart contracts to hold funds in escrow until predefined milestones are completed and approved.

The element in scope for this audit, Trustless Work's Smart Escrow, implements the core functionalities related to the management of assets, milestones, and general features regarding agreements between the different actors involved throughout its lifecycle.

Smart Escrow

At its core, Trustless Work's contracts intermediate the agreements by enabling users to instantiate new smart escrows with specific users registered with authority roles over an agreement. This agreement is formed virtually between two parties: one can be seen as a buyer of a service or product, while the other is a service provider or product seller. The escrow holds the buyer's funds until there's confirmation from all involved parties that the service has been fulfilled or that the product has been properly delivered. Then, the escrow sends them to the seller/service provider once all approvals are met. If these approvals are not met, users can raise disputes to reclaim their deposited assets, with the help of the authorities involved in this agreement/negotiation.

The authorities, also referred to as actors, are described, according to [Trustless Work's documentation](#) and our discoveries during the design review stage, as follows:

- Approver: Address of the entity requiring the service, or buying the product.
- Service Provider: Address of the entity providing the service, or selling the product.
- Platform Address: Address of the entity that owns the escrow, which can be seen as the platform where the negotiation/agreement takes place.
- Release Signer: Address of the user in charge of releasing the escrow funds to the service provider.
- Dispute Resolver: Address in charge of resolving disputes within the escrow.
- Receiver: Address to which escrow proceeds will be sent.
- Funder: Although not explicitly mentioned in Smart Escrow's documentation, the funder is another entity expected to appear in an escrow's lifetime. This entity is responsible for funding the escrow. Anyone can be a funder, and an escrow may have one or more funders. Since



anyone can be a funder, there are no validations over address related to the funding process of a contract.

Additionally, all roles are capable of opening disputes in case there's a disagreement regarding the fulfillment of the milestones or the original agreement in general.

The roles above are represented throughout the smart escrow code with similar terminology, often following standard coding conventions (no spaces, camel case).

From a lower-level perspective, the Smart Escrow comprises a constructor and 12 endpoints. The constructor takes an `admin` address parameter, which is stored in instance storage.

The core endpoints for the business logic of the Trustless Work system are described below:

- `initialize_escrow` : initializes the Smart Escrow with the desired configuration. Takes an `Escrow` struct, representing the properties of the smart escrow (roles, milestones, fees, flags, etc.) as a parameter. These properties are stored in the contract's instance storage and are used to validate if the escrow has already been initialized before. Additionally, to be initialized, an escrow must have at most 10 milestones, and the amount to be processed by it must be greater than 0.
- `fund_escrow` : used to transfer assets from the buyer to the escrow. Receives a signer address and an amount to fund the escrow as parameters. There are no validations over who the signer is (other than the caller itself) and the amount to be deposited. The smart contract then proceeds to get the trustline, which is the chosen token to represent the payment asset in the agreement, and transfers it from the caller to the escrow.
- `release_funds` : used to transfer the assets to their rightful owners after the confirmation of completion of services. This endpoint takes the release signer and the trustless work platform addresses as parameters. In this endpoint, the state of the escrow is validated to ensure that the assets can be released (escrow `released` flag is set to `true`), that the caller address is indeed the release signer, that the escrow has milestones and that they are all approved, and that the escrow is not flagged as disputed. If these conditions are respected and the escrow has enough funds, the Trustless Work fee address and the platform address receive their fees, and the `receiver` receives payment for the provided services.
- `approve_milestone` : used by the approver to agree that a specific milestone has been achieved. Takes the milestone index, the new status flag, and the approver address as parameters. It is enforced that the caller must be the `approver` stored in the escrow properties, that this escrow has milestones, and the index of the milestone being approved is



within the bounds supported by the escrow. If these conditions are respected, the escrow properties in storage are modified to reflect the change in the milestone state (approved flag).

- `change_milestone_status` : used by the service provider to configure the status of an individual milestone. Takes the milestone index, a status string, an optional string representing the evidence of status change, and the service provider address as parameters. It is enforced that the caller must be the `service_provider` stored in the escrow properties, that this escrow has milestones, and the index of the milestone being approved is within the bounds supported by the escrow. If these conditions are respected, the escrow properties in storage are modified to reflect the change in the milestone state (status and proof string).
- `dispute_escrow` : can be used by any actor configured in the escrow properties to change the status of the escrow to disputed, preventing the full payment of the service provider in case of failure to fulfill their end of the agreement. Receives a signer address as a parameter. This endpoint verifies that the escrow is not already in a disputed state and that the caller is one of the addresses configured in escrow property roles. If these requirements are met, the escrow disputed flag is changed to `true`.
- `resolve_dispute` : In case of a dispute, the configured dispute resolver must use this endpoint to intervene and decide how to split the funds held in the escrow between the receiver and approver. It receives the address of the dispute resolver as a parameter, the amount of funds that should be owned by the approver, the amount of funds that should be owned by the receiver, and the trustless work platform address. It enforces that the caller must have the same address as the stored dispute resolver, that the contract must have enough balance to pay both the approver and the receiver. Once the fees for the platform and for the Trustless Work entities are calculated and transferred, the approver and receiver are paid. The escrow status flags are modified (`resolved` set to `true`, and `disputed` set to `false`).

Although not necessary to its proper functioning, the escrows provide an `update_escrow` endpoint. It is used by the platform address to modify the escrow properties using the same structure as the one provided in the escrow initialization. This endpoint only works successfully if the caller is the platform address, the escrow hasn't been disputed, if its milestones are not approved, and the escrow holds no assets. The platform address cannot be modified. Essentially, these changes can only be performed at the beginning of the lifetime of a Smart Escrow.

Additionally, Smart Escrow has three endpoints that can be used to query general escrow information: `get_escrow`, `get_escrow_by_contract_id`, and `get_multiple_escrow_balances`.

Finally, the last endpoint is `deploy`. This function is used to deploy and initialize contracts through the smart escrow. It takes as parameters the deployer address, a contract wasm hash, a salt (nonce), the symbol of the function for the contract initialization, and the contract initialization parameters. Additionally, the specified initialization function of the deployed contract is also called, preventing front-running attacks from happening.

Smart Escrow Lifetime and Operations

In its normal flow of operations, the smart escrow is deployed, funded, its milestone statuses are set by the service provider, gradually approved by the approver, and once all of the milestones are met, the funds held at the escrow can be released, paying the fees to the involved entities (platform and Trustless Work) and the recipient. This flow of actions can be visualized in Figure 1. The transfers performed after the `release_funds` operation are triggered as a consequence of it.

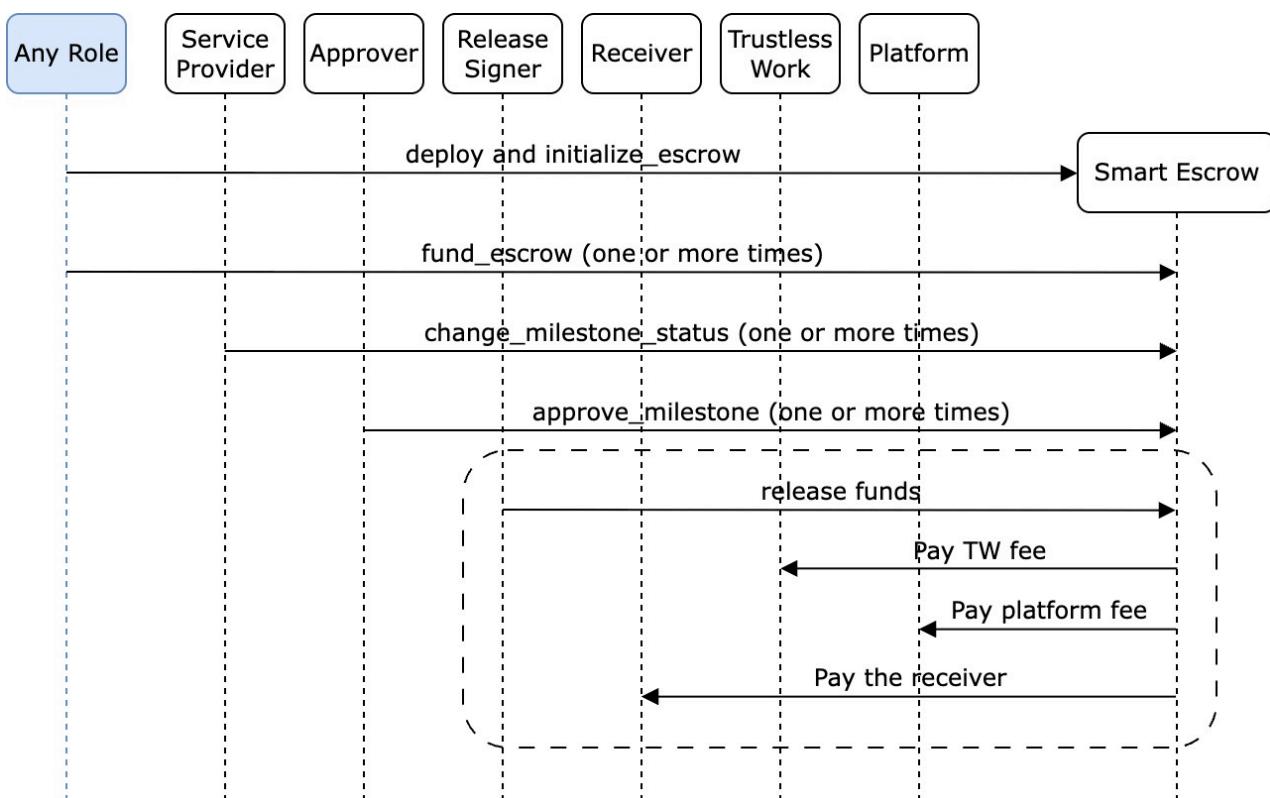


Figure 1: Ideal lifetime of a Smart Escrow.

Once initialized, at any point in its lifetime, the Smart Escrow can be disputed. Once in the disputed state, only the designated dispute resolver is capable of modifying the disputed flag in the escrow

state. This authority figure has complete control over how the funds in the Smart Escrow will be divided between the Approver and Receiver, as well as the address representing the Trustless Work fee receiver. The `resolve_dispute` operation internally triggers the fee payments as well as the payments for the Approver and Receiver, as shown in Figure 2.

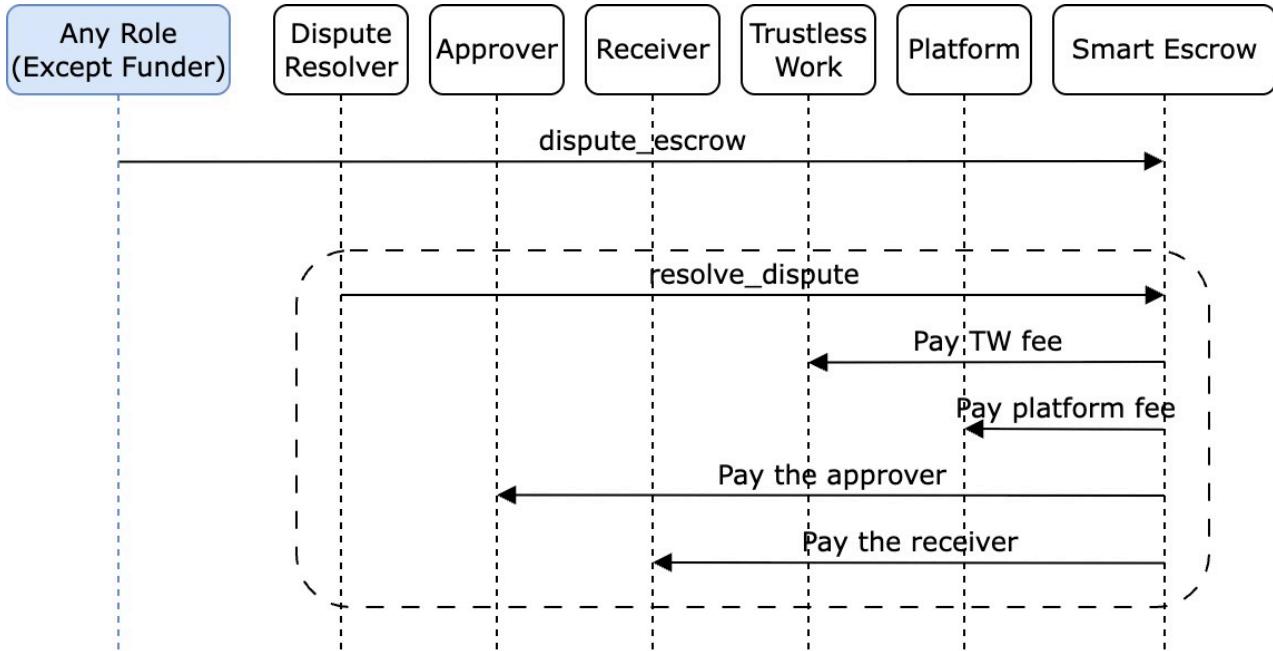


Figure 2: Diagram of actions performed in case of dispute in the Smart Escrow.

Multi-Release Version Extension

As an extension of the engagement, we also explored the Multi-Release Version of the Smart Escrow. This version has the capability of performing the vesting of assets on a per-milestone basis. The amount to be vested in the Smart Escrow, as well as the flags defining its status are not contained in the `Escrow struct` anymore, instead being held in the `Milestone struct`.

Additionally, the management of the status of a Smart Escrow, as well as any modifications related to the vesting amounts, is performed at a milestone level, conceptually modifying `release_funds`, `dispute_escrow`, and `resolve_dispute`. As a consequence, these functions are replaced by `release_milestone_funds`, `dispute_milestone`, `resolve_milestone_dispute`. The new functions have the same logic as their Single-Release counterparts, previously elaborated on in the Smart Escrow section above, but now receive the milestone index as a parameter to identify the milestone the operation refers to.

Trustless Work's Backend

The Trustless Work Backend, the target of a design review during this security engagement, is a NestJS-based API service that acts as the middleware layer between client applications and the Stellar blockchain infrastructure. It provides a RESTful interface for managing smart escrow contracts, handling blockchain transactions, and maintaining off-chain data persistence through Firebase Firestore.

Core Architecture Components

The backend currently implements a modular microservices-style architecture with the following key modules:

- **Authentication Module:** Provides JWT-based authentication and authorization using Passport strategies.
- **Stellar Contract Module:** Core business logic for escrow operations, transaction building, and blockchain interactions.
- **Firebase Module:** Data persistence layer using Firestore for escrow metadata, user information, and transaction history.
- **Indexer Module:** Blockchain event monitoring and data synchronization services.
- **Notifications Module:** Real-time notification system for escrow status updates.
- **Queue Module:** In-memory transaction queue management for pending blockchain operations.

Backend Service Architecture

The backend implements a service-oriented architecture with specialized services organized by functional responsibility:

Authentication Services (`auth.service.ts`):

- User registration and login management.
- JWT token generation and validation.
- Wallet-based authentication integration with Firebase.

User Management Services (`user.service.ts`):

- User profile creation and updates.
- API key management and association.

- User data retrieval and management operations.

Firebase Services (`firebase.service.ts`):

- Firestore database connection and operations.
- Document management and querying.
- Real-time data synchronization.

Stellar Contract Services:

- **Transaction Builder Service (`stellar-transaction-builder.service.ts`)**: Constructs unsigned XDR transactions for various escrow operations (deploy, fund, release, dispute resolution).
- **Helper Service (`helper.service.ts`)**: Handles signed transaction submission to the Stellar network, processes responses, and manages escrow queries.
- **Deployer Service (`deployer.service.ts`)**: Manages deployment of new escrow contracts with proper initialization parameters.
- **Shared Escrow Service (`shared-escrow.service.ts`)**: Provides common escrow operations shared between single and multi-release contracts.
- **Single/Multi Release Services (`single-release.service.ts` , `multi-release.service.ts`)**: Specialized business logic for respective escrow types.

Queue Management Services:

- **Pending Write Queue Service (`pending-write-queue.service.ts`)**: In-memory queue management for pending blockchain operations.
- **Pending Write Handler Service (`pending-write-handler.service.ts`)**: Processes queue items and updates database state after successful transactions.

Data Services:

- **Escrow Firestore Service (`escrow-firebase.service.ts`)**: Specialized Firestore operations for escrow data management.
- **Indexer Service (`indexer.service.ts`)**: Blockchain event monitoring and data synchronization.
- **Notifications Service (`notifications.service.ts`)**: Real-time notification system for escrow status updates.

Contract Template Services (`contract-templates.service.ts`):

- WASM contract template management and deployment configurations.

API Interface

The backend currently exposes RESTful endpoints organized into the following controller modules:

Authentication Controller (`auth.controller.ts`):

- Request API keys for wallet-based authentication.
- JWT token generation and validation.

User Management Controller (`user.controller.ts`):

- Create new user profiles.
- Update user information and wallet addresses.
- Manage API key associations.
- Retrieve user details and list all users.

Deployer Controller (`deployer.controller.ts`):

- Deploy new single-release escrow contracts.
- Deploy new multi-release escrow contracts with milestone-based vesting.

Single Release Escrow Controller (`single-release.controller.ts`):

- Fund escrow with USDC tokens.
- Approve milestone completion.
- Release funds to the service provider.
- Update escrow properties.
- Initiate and resolve dispute processes.

Multi Release Escrow Controller (`multi-release.controller.ts`):

- Fund multi-release escrow contracts.
- Approve individual milestones.
- Release funds for specific milestones.
- Update milestone-specific properties.
- Dispute and resolve individual milestones.

Helper Controller (`helper.controller.ts`):

- Submit signed transactions to the blockchain.

- Retrieve escrows associated with user wallet addresses.
- Query escrows by participant role (approver, service provider, etc.).
- Fetch specific escrow details by contract identifier.
- Establish trustlines for USDC token operations.
- Send funds with a memo for external transfers.

Indexer Controller (`indexer.controller.ts`):

- Update escrow data from transaction hashes.
- Synchronize blockchain state with database records.

Notifications Controller (`notifications.controller.ts`):

- Test notification system functionality.
- Manually trigger checks for pending escrows, disputes, inactive escrows, and completed milestones.

Data Management and Persistence

The backend maintains a hybrid storage model:

1. **On-Chain Data:** Core escrow logic, milestone states, role assignments, and fund balances stored in Stellar smart contracts.
2. **Off-Chain Data:** Metadata, user profiles, transaction history, and cached contract states stored in Firestore for faster query performance.
3. **Data Consistency:** Built-in validation mechanisms ensure synchronization between blockchain state and database records.

Transaction Flow Management

The system implements a sophisticated transaction queue mechanism:

1. **Queue System:** `PendingWriteQueueService` manages pending blockchain operations with metadata.
2. **Transaction Building:** Services construct unsigned transactions with appropriate parameters and queue actions.
3. **Client Signing:** Frontend applications sign transactions using wallet integrations (Freighter, etc.).

- 
4. **Submission and Processing:** Backend submits signed transactions and processes results asynchronously.
 5. **State Updates:** Successful transactions trigger database updates and notification events.

Authentication and Security

- JWT-based authentication with Passport integration.
- Bearer token authorization for protected endpoints.
- Rate limiting through NestJS Throttler (50 requests per minute).
- Input validation using class-validator decorators.
- CORS configuration for cross-origin requests.

External Dependencies

- **Stellar SDK:** Blockchain interaction and transaction management.
- **Firebase Admin:** Firestore database operations and authentication.
- **Swagger/OpenAPI:** API documentation and client generation.
- **Development Tools:** ESLint, Prettier, Husky for code quality.

Client Integration

Client applications interact with the backend through a typical flow:

1. **Authentication:** Obtain JWT tokens through wallet-based authentication.
2. **Escrow Creation:** Submit escrow parameters to deployment endpoints.
3. **Transaction Signing:** Receive unsigned XDR transactions for client-side signing.
4. **Transaction Submission:** Return signed transactions to completion endpoints.
5. **State Monitoring:** Query escrow status and receive real-time notifications.

The backend abstracts blockchain complexity while providing developers with familiar REST API patterns, enabling rapid integration of escrow functionality into web and mobile applications.

Invariants

During the audit, invariants were defined and used to guide part of our search for possible issues with Trustless Work's Smart Escrows. Using the previously explored specifications, the client's documentation, the intended business logic, and references collected during the audit, we identified the following invariants for the single-release smart escrow:

General

- Trustline must be set on escrow contract
 - Failure to set a trustline must not result in loss of funds or broken smart escrow functionality
 - Trustline must be a valid Stellar Asset
- All funds that are paid out (either released or refunded) pay relative fees:
 - Fees to infrastructure provider -> Trustless Work
 - Platform for dispute resolution -> Platform
- The Smart Escrow must be initialized at contract creation time
- The Smart Escrow can never be initialized more than once
- These Smart Escrow properties can only be changed by the platform as long as no funds have been stored yet, no milestones have been approved, and no dispute is present, otherwise they are immutable:
 - Infrastructure provider fee
 - Platform fee
 - Trustline
 - Amount
 - All addresses of the specified roles
 - Milestones:
 - Description
- The Trustless Work address and fee, specified in BPS (basis points), must be immutable.
- All milestones must be initialized to:
 - checked: empty status and empty proof
 - approved: false

Roles

- `approver` :
 - Milestones that have been checked can be approved only by the `approver` .
 - The `approver` receives the refunded funds minus the respective relative fees in case of a resolved dispute.
- `service_provider` :
 - Only the `service_provider` can change the checked status (status: String, proof: Optional).
- `platform_address` :
 - Only the `platform_address` can change the Smart Escrow properties with the same limitations as when initializing the Smart Escrow as long as no funds have been stored yet, no milestones have been approved, and no dispute is present
 - Only the `platform_address` receives platform fees when either funds are sent to `receiver` or `approver` during funds release or dispute resolution. The platform fees that are specified by `platform_fee_bps` in bps (basis points, 1 BPS = 0.01%) are relative to the funds sent, which in total amount to `amount` .
- `release_signer` :
 - Only the `release_signer` can sign the transaction that sends the funds of the Smart Escrow minus the fees to the `receiver` after all milestones have been approved and no dispute was raised. The respective fees are sent to Trustless Work and the `platform_address` in the same transaction call.
- `dispute_resolver` :
 - Only the `dispute_resolver` can resolve a raised dispute. The `dispute_resolver` decides if and how much of the specified funds, `amount` , are refunded to the `approver` . The non-refunded funds minus the respective relative fees are sent to `receiver` .
- `receiver` :
 - After checking all milestones, successful approval of all milestones, no un-resolved dispute, sign-off by `releaseSigner` , and having the specified `amount` as funds available in the Smart Escrow, the funds specified in `amount` must be transferred to `receiver` minus the Trustless Work fees and Platform fees.

- In case of a dispute, the `receiver` receives none, a part of, or the entirety of the Smart Escrow funds specified by `amount` minus the respective relative fees. The exact amount sent to `receiver` minus the fees is specified by the `dispute_resolver` in a resolve dispute transaction.
- Trustless Work - infrastructure provider:
 - For providing infrastructure outside the Smart Escrow smart contract, a fee akin to the platform fee is paid out to Trustless Work. This fee is specified in bps (basis points) at smart contract compile time. Trustless Work enforces that the very revision of the Smart Escrow is used at runtime. None of the roles must be able to redirect the Trustless Work fees to another address other than specified by Trustless Work themselves.
- `approver`, `service_provider`, `platform_address`, `release_signer`, `dispute_resolver`, and `receiver`
 - All of the available roles can dispute the Smart Escrow as long as the funds were not paid out yet. A Smart Escrow can not be disputed a second time.

Status state changes

Invariants of status state changes with linear temporal logic (LTL). The status flags reflect the abstract states of the Smart Escrow, not necessarily the internal flags from the implementation.

- When the funds are released, the balance of the Smart Escrow is decreased by the specified `amount` :
 - $\square(\neg\text{released} \wedge \bigcirc(\text{released}) \rightarrow \text{balance} - \text{amount} = \text{newbalance}; \text{with } \bigcirc(\text{balance} = \text{newbalance}))$
- when a dispute is resolved, the entire funds of the Smart Escrow (`balance`) must be transferred:
 - $\square(\text{disputed} \wedge \bigcirc(\neg\text{disputed}) \rightarrow \bigcirc(\text{balance} = 0))$
- if the funds have not been released, the Smart Escrow has been disputed, and the dispute has not yet been resolved, then the funds cannot yet be released:
 - $\square(\neg\text{released} \wedge \neg\text{disputed} \rightarrow \bigcirc(\neg\text{released}))$
- if any milestone is not approved, then the funds cannot be released
 - $\square(\bigvee_{m \in \text{milestones}} (\neg m.\text{approved}) \rightarrow \neg\text{released})$



Funds

- Each Smart Escrow manages one set of funds. These funds can be transferred to the Smart Escrow by different individuals with multiple transactions.
- Using the Smart Escrow requires paying fees to:
 - a) Trustless Work - for providing Smart Escrow Infrastructure
 - b) Platform
- The fees are paid out exactly when funds are released from the Smart Escrow in the following functions: `release_funds` and `resolve_dispute`:
 - For `release_funds`, the following must always hold true:

```
total_released_funds = sent_to_receiver + fees_sent_to_trustless_work + fees_sent_to_platform =
(1 - (trustless_work_fees_bps + platform_fees_bps)/BPS_DENOMINATOR) * amount +
(trustless_work_fees_bps/BPS_DENOMINATOR) * amount + (platform_fees_bps/BPS_DENOMINATOR) *
amount = amount where BPS_DENOMINATOR = 10000
```

- For `resolve_dispute`, the funds that are sent out in total must equal the available balance of the Smart Escrow, which might be a number higher, equal, or lower than `amount`. In total, funds might be sent to either the `receiver`, `approver`, or both, excluding the fees that are paid to Trustless Work and `platform`. Therefore, for `resolve_dispute`, the following must always hold true:

```
total_released_funds = sent_to_receiver + sent_to_approver +
fees_sent_to_trustless_work_for_receiver + fees_sent_to_trustless_work_for_approver +
fees_sent_to_platform_for_receiver + fees_sent_to_platform_for_approver =
sent_to_receiver_and_approver + fees_sent_to_trustless_work + fees_sent_to_platform = (1 -
(trustless_work_fees_bps + platform_fees_bps)/BPS_DENOMINATOR) * balance +
(trustless_work_fees_bps/BPS_DENOMINATOR) * balance + (platform_fees_bps/BPS_DENOMINATOR) *
balance = balance
```

- After a `release_funds`, the balance must hold `total_of_all_escrow_deposits - amount` where `total_of_all_escrow_deposits` is the total sum of all funds ever deposited to this Smart Escrow.
- After a `resolve_dispute`, the balance must be `0`.
- The total fees can never exceed 100%, therefore the following must always hold true:
`trustless_work_fee_bps + platform_fee_bps <= 1`.



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).

[A01] Dispute resolver and platform can call with arbitrary `trustless_work_address` and redirect TW fees to another address

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

Some public Smart Escrow functions accept the parameter `trustless_work_address: Address`, which is the address where the Smart Escrow sends the Trustless Work fees. The two functions that accept this parameter are `resolve_dispute` and `release_funds`, both of which do not validate the address. This means that the caller of the functions can send an arbitrary address instead of the actual Trustless Work address, thereby allowing the redirection of Trustless Work fees to an unintended recipient.

However, both `resolve_dispute` and `release_funds` are access-restricted. In the case of `release_funds`, only the `release_signer` can sign the function call, while for `resolve_dispute`, only the `dispute_resolver` can sign the function call.

Recommendation

We recommend hard-coding the address of Trustless Work into the Smart Contract or using some other secure method to query the address at runtime.

Status

The client has addressed this finding by hard-coding the Trustless Work address into the contract and using it internally instead of requiring it as an input parameter. The modification was already implemented in another version of the contract, which was deployed for testing.

[A02] Maximum relative amount of fees not checked at initialization time

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

The Smart Escrow initialization function does not check whether the fees exceed a maximum amount.

There are two kinds of fees: the `trustless_work_fee` and the `platform_fee`. The `trustless_work_fee` is set to a fixed value at Smart Contract compilation time, while the `platform_fee` is passed as a parameter at Smart Contract initialization time. Both fees are relative to the total amount of funds managed by the Smart Escrow. The fees are specified in the unit BPS, which stands for basis points with one unit being equal to 0.01%.

In case the sum of both fees exceeds 100%, the funds will be locked in the Smart Escrow.

This is due to the Smart Contract attempting a transfer with a negative amount after deducting a relative fee of over 100%. Typically, token implementations revert on transfers with negative amounts. In this case, the funds will be locked in the Smart Escrow. E.g., for `resolve_dispute`, the following code computes the fees and net amount for `approver`:

```
let trustless_work_fee = SafeMath::safe_mul_div(
    total_resolved_funds,
    TRUSTLESS_WORK_FEE_BPS,
    BASIS_POINTS_DENOMINATOR,
)?;
let platform_fee = SafeMath::safe_mul_div(
    total_resolved_funds,
    platform_fee_bps,
    BASIS_POINTS_DENOMINATOR,
)?;
let total_fees = BasicMath::safe_add(trustless_work_fee, platform_fee)?;
let net_approver_funds = if total_resolved_funds > 0 {
    let approver_fee_share =
        SafeMath::safe_mul_div(approver_funds, total_fees, total_resolved_funds)?;
    BasicMath::safe_sub(approver_funds, approver_fee_share)?
} else {
```

```
0  
};
```

Here, the variable `net_approver_funds` will be negative in case `total_fees` is bigger than `total_resolved_funds`.

Recommendation

We recommend that during both initialization and Smart Escrow update, the sum of the `trustless_work_fee` and `platform_fee` is validated to be less than 100%.

Status

This finding has been addressed in commit ID `1da344e` in branch `single-release-develop`.

[A03] Initialization possible with pre-approved milestones

Severity: Medium

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

At initialization, an `Escrow` object is passed that defines the Smart Escrow properties, including the details of the milestones. For this purpose, the passed `Escrow` struct is validated with the function `validate_initialize_escrow_conditions`.

This function does not check whether any passed milestone already has the approved flag set to true. This allows the contract initializer to circumvent the `approver` role.

Recommendation

We recommend to either:

- validate during initialization and smart escrow update that the passed `Escrow` object has all `approved` flags set to false
- override during initialization and smart escrow update the passed `Escrow` object and set all `approved` flags to false

Status

This finding has been partially addressed in commit ID `6d979b5` in branch `single-release-develop`.

[A04] Bypass of the Escrow Initialization Validations

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Partially addressed by client

Description

At initialization, an `Escrow` struct is passed that defines the Smart Escrow properties. For this purpose, the passed `Escrow` struct is validated with the function

`validate_initialize_escrow_conditions`. When initializing an instance of a Smart Escrow, using the `initialize_escrow` endpoint, prior to definition of the escrow state, the contract business logic enforces that it is initialized before, that the payment amount of the escrow is not zero, and that the escrow cannot have more than 10 milestones.

Additionally, the `platform` role may update the Escrow properties with the public function `update_escrow`, as long as no funds have been stored yet, no milestones have been approved, and no dispute is present. This is similarly done by passing a new `Escrow` struct that defines the new Smart Escrow properties.

However, in contrast to the initialization, no validation is performed on the passed `Escrow` struct. Notice that, even though the initialization endpoint and the update endpoint have extremely similar purposes (modifying the escrow state), the set of validations performed by each endpoint is different. This creates opportunities for unpredictable behaviors, as, for instance, an escrow can be updated to have more than 10 milestones, even though its initialization enforces that this shouldn't be possible. Similarly, an escrow can be updated for its amount to be zero, even though its initialization enforces that this shouldn't be possible.

Recommendation

We recommend to use the same function `validate_initialize_escrow_conditions` that is used during initialization to validate the Escrow properties in the `update_escrow` function.

Status

This finding has been partially addressed in finding [6d979b5](#) in branch `single-release-develop`. `validate_escrow_property_change_conditions` and `validate_initialize_escrow_conditions`



still have distinct implementations.

[A05] Front-running attack by a malicious platform on fund_escrow with update_escrow

Severity: High

Difficulty: High

Recommended Action: Fix Code

Addressed by client

Description

A user can fund the Smart Escrow by utilizing the Smart Escrow public function `fund_escrow`. For this purpose, the user only needs to specify the amount of tokens to deposit.

The `platform` role may change the Smart Escrow properties, provided no funds have been deposited, no milestones approved, and no dispute is ongoing. This can be done by executing the Smart Escrow public function `update_escrow`.

This allows a malicious `platform` actor to try to front-run a pending user transaction that executes the `fund_escrow` with an `update_escrow` transaction. This would result in a user funding a Smart Escrow that has been changed since the `fund_escrow` has been signed.

In particular, the changeable properties include the roles, meaning the malicious `platform` actor could change the roles and subsequently steal the funds.

Recommendation

Require the user to specify the `Escrow` struct as a parameter in `fund_escrow` and validate in `fund_escrow` that the passed `Escrow` struct matches the stored `Escrow` struct. In case of a mismatch, revert, as the user might not want to fund a Smart Escrow with different properties.

Status

This finding has been addressed in commit ID `6d979b5` in branch `single-release-develop`.

[A06] Smart Escrow uses signed integer type `i128` without sign checks

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Partially addressed by client

Description

The Smart Escrow uses the `i128` type for multiple attributes, but is missing sign checks at different locations. This allows for negative parameters to be passed to Smart Escrow public functions and Smart Escrow attributes. On this topic, we highlight:

1. `platform_fee`

In particular, the Smart Escrow uses `i128` for the constant `TRUSTLESS_WORK_FEE_BPS` and the Smart Escrow attribute `platform_fee`. While `TRUSTLESS_WORK_FEE_BPS` is a constant and guaranteed to be positive, `platform_fee` is initialized by `initialize_escrow` or `update_escrow`. While `update_escrow` is currently missing validation for escrow properties entirely, as outlined in [\[A04\] Bypass of the Escrow Initialization Validations](#), initialization does validate properties generally. However, it is not checked at initialization time whether the `platform_fee` is negative.

This means that a Smart Contract user can specify a negative `platform_fee` that cannot be changed anymore once funds are transferred to the Smart Escrow. In this case, **the funds would be locked** in the Smart Escrow, as both `release_funds` and `resolve_dispute` would revert when attempting to transfer the negative amount of relative platform fees.

2. `fund_escrow` parameter `amount_to_deposit`

Typically, tokens revert on a negative `amount` parameter, but this behavior might vary as Soroban tokens specifically use `i128`, in case a token wants to implement functionality for negative balances and similar. Therefore, this revert behavior of tokens should not be relied upon.

The Smart Escrow public function `fund_escrow` accepts the parameter `amount_to_deposit: i128` and passes that value to `transfer` of a token specified in the Smart Escrow attribute `trustline`. As previously outlined, the Smart Escrow should not rely on tokens reverting to negative numbers and should check whether the given `amount_to_deposit` is negative.

3. `resolve_dispute` allows negative funds parameters



The Smart Escrow public function `resolve_dispute` allows negative values for both `approver_funds` and `receiver_funds`. While it is checked that the sum `approver_funds + receiver_funds` is equal to the balance of the Smart Escrow, this check does not cover the case where one of the parameters is negative, while the other is larger than `balance`, such that the sum still equals the Smart Escrow `balance`.

4. `amount`

The Smart Escrow attribute `amount: i128` specifies the amount to be released in `release_funds` and is initialized in either `initialize_escrow` or `update_escrow`. While `update_escrow` is currently missing validation for escrow properties entirely, as outlined in [\[A04\] Bypass of the Escrow Initialization Validations](#), initialization does validate properties generally. Though it is not checked at initialization time whether the `amount` is negative.

This means that a negative `amount` attribute could be specified during initialization that cannot be changed later on once funds have been transferred to the Smart Escrow. A negative `amount` attribute can cause `release_funds` to revert.

5. `milestone_index` parameter

The Smart Escrow uses the signed integer type `i128` as the type for the `milestone_index` parameter for multiple contract functions. While there are either explicit or implicit bound checks in place, negative milestone indices might overflow when cast to an unsigned integer.

Recommendations

For handling the discussed topics above, our recommendations are, respectively:

1. We suggest that validation of the `platform_fee` attribute during initialization and escrow update prevents negative values. Furthermore, we suggest that the type of `platform_fee` and `TRUSTLESS_WORK_FEE_BPS` be changed to a smaller unsigned integer type, such as `u16`, as the fee attributes have a maximum value as well, which is expected to be less than 10,000 and can therefore be safely represented by a `u16`.
2. We suggest that a check is added in `fund_escrow` that validates that `amount_to_deposit` is greater than zero.
3. Add additional checks in `resolve_dispute` that validate that both `approver_funds` and `receiver_funds` are non-negative.

- 
4. We suggest that validation of the `amount` attribute during initialization and escrow update prevents negative values.
 5. Change the signed integer type to an unsigned integer type for `milestone_index`.
-

Status

This finding has been partially addressed in commit IDs `6d979b5`, `58539ca`, and `33bfaf8` in branch `single-release-develop`. Regarding topic 4, the approach used to address the issue was to implement an enforcement that the funding amount should be greater than or equal to 0 when handling escrow via the `fund` endpoint. Although this is effective for handling the escrow in its intended usage, an escrow can still be initialized with a negative amount. To prevent issues related to this altogether, all instances where `Escrow.amount` is used should be validated against negative values, or more simply, the `Escrow` struct should use an unsigned type for `amount`.

[A07] The Approver Can Disapprove Milestones

Severity: Low

Difficulty: Low

Recommended Action: Fix Design

Addressed by client

Description

The Smart Escrow contract contains a design flaw that exposes it to a griefing attack by the Approver. While the Approver's role is to approve milestones, the `approve_milestone` function allows them to both approve and disapprove milestones using a boolean flag, defined by the `new_flag` parameter. This is a problem because it enables the Approver to maliciously change the state of an already approved milestone, effectively reverting its status and disrupting the entire escrow process. This behavior does not result in any direct financial gain for the Approver but causes significant harm to other parties, particularly the service provider, by hindering their progress and preventing funds from being released.

While this is not a critical vulnerability resulting in direct fund loss, it remains a serious design flaw. A griefing attack is where an actor uses a system's intended features to disrupt without gaining a direct financial advantage. By disapproving a previously approved milestone, the Approver can stall the project indefinitely, causing frustration, delaying payments, and, in the case of a multi-source funds escrow, potentially causing the entire crowdsourcing effort to fail. Since the protocol's lifecycle depends on milestones being approved, the ability to revert a milestone's status creates a single point of failure and a denial of service vector for the contract's functionality, which still can be handled by forcing a dispute.

Recommendation

We recommend modifying the `approve_milestone` function to prevent this type of attack by removing the `new_flag` parameter or enforcing that it is always set to `true`. Design-wise, suppose a legitimate Approver notices a problem with a previously approved milestone. In that case, they can still invoke a dispute, forcing the intervention of an authority that can investigate the issue.

Status

This finding has been addressed commit IDs [fe2623f](#) and [2b31f1e](#) in branch [single-release-develop](#).

[A08] Exceeding Assets Will Be Permanently Locked in The Multi-Release Smart Escrows

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

When dealing with the release of its funds, the Single-Release version of the Smart Escrow expects a pre-determined amount to be withdrawn via the `release_funds` endpoint or the total balance of the escrow via the `resolve_dispute` endpoint. If the escrow holds an excess balance beyond the expected amount, the dispute mechanism can be used to handle it.

For the Multi-Release Smart Escrow, funds are released on a milestone basis, with pre-determined amounts per milestone. The dispute method, which is also per milestone, ensures that the disputed value cannot be greater than the pre-determined amount to be released for that milestone. This means that the sum of values released via hypothetical disputes is the same as the amount released through the proper release method, indicating that, if the balance of the Multi-Release Smart Escrow is bigger than the amount expected to be released, the exceeding assets cannot be withdrawn. These tokens will be permanently locked in the contract.

The caveat here comes from finding [\[A09\] Milestones can both be released and dispute-resolved in the multi-release smart escrows](#), which considers that a milestone can both be released and disputed, meaning that, although incorrectly, the redeemable exceeding balance of a Multi-Release Smart Escrow, in its current state, can be at most the same as its expected release value.

Recommendation

The Multi-Release Smart Escrow should implement an additional endpoint exclusively for the `dispute_resolver`, which works similarly to `resolve_dispute` in the Single-Release escrow. This endpoint should work on the condition that all milestones must have either been released or dispute-resolved. Alternatively, the limitations on the withdrawn amounts can be removed, but this introduces the issue of being able to withdraw the full balance of an escrow through a single milestone dispute.

Status

This finding has been addressed in commit ID `2f6ef92` in branches `multi-receiver`, `multi-release-develop`, `multi-release-main`, and `stellar-multi-release-audit`.

[A09] Milestones can both be released and dispute-resolved in the multi-release smart escrows

Severity: Medium

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

In the multi-release smart escrows, amounts are released on a per-milestone basis. Each milestone defines a specific amount to be released upon completion. At the same time, the smart contract limits the amount that can be released during a milestone dispute resolution to at most that value specified in the respective milestone.

There are no mechanisms in place that disallow a milestone dispute resolution after a milestone has been released. Additionally, there are also no mechanisms that disallow a milestone release after a milestone dispute resolution. Therefore, up to twice the amount specified per milestones can be released per milestone, draining the funds of the smart escrow that are allocated for other milestones.

In the single-release smart escrow, this is a deliberate design decision that allows withdrawal of exceeding funds. Though this design is not suited for a smart escrow with multiple releases of funds.

Recommendation

Add validations that enforce that dispute-resolved milestones cannot be released as well as that released milestones cannot be dispute-resolved.

There is another issue that can cause funds to be permanently locked in the multi-release smart escrows, described in [\[A08\] Exceeding Assets Will Be Permanently Locked in The Multi-Release Smart Escrows](#). The recommended fix reduces the amount of funds that can be recovered from the smart escrow when locked by the issue described in [\[A08\] Exceeding Assets Will Be Permanently Locked in The Multi-Release Smart Escrows](#). Therefore it is heavily recommended to implement the recommendations from finding [\[A08\] Exceeding Assets Will Be Permanently Locked in The Multi-Release Smart Escrows](#).



Status

This finding has been addressed in commit ID `bbdfcc4` in branch `multi-receiver`,
`multi-release-develop`, `multi-release-main`, and `stellar-multi-release-audit`.



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.

[B01] Gas optimizations

Severity: Informative

Recommended Action: Fix Code

Addressed by client

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

`token_client.balance(&contract_address)` called twice in `core/dispute.rs:46`

The gas costs can be optimized by storing the result of the first invocation of `token_client.balance(&contract_address)` in `core/dispute.rs:46` and replacing the second invocation with the cached result.

```
if token_client.balance(&contract_address) < total_funds {  
    return Err(ContractError::InsufficientFundsForResolution);  
}  
  
let fee_result = FeeCalculator::calculate_dispute_fees(  
    approver_funds,  
    receiver_funds,  
    escrow.platform_fee as i128,  
    total_funds,  
)?;  
  
let current_balance = token_client.balance(&contract_address);
```

Redundant out-of-bounds check of milestone index

The function `change_milestone_status` uses the function `validate_milestone_status_change_conditions` to validate some properties, including that the index is within bounds, see `validators/milestone.rs:19-21`:

```
if milestone_index < 0 || milestone_index >= escrow.milestones.len() as i128 {  
    return Err(ContractError::InvalidMileStoneIndex);  
}
```

This check is unnecessary, as the property is already validated in the function

`change_milestone_status`, see `core/milestone.rs:30-33`:

```
let mut milestone_to_update = existing_escrow
    .milestones
    .get(milestone_index as u32)
    .ok_or(ContractError::InvalidMileStoneIndex)?;
```

The function `change_milestone_status` is the only function that calls the validation function `validate_milestone_status_change_conditions`. Afterwards, the parameter `milestone_index` can be removed as well.

Similarly, the functions `change_milestone_approved_flag` and `validate_milestone_flag_change_conditions` can be optimized in the same manner, see `validators/milestone.rs:40-42` and `core/milestone.rs:62-65`.

Empty milestones checks outside of Escrow initialization and update

Currently, it is not checked at initialization (see [\[B02\] Initialization possible with zero milestones](#)) whether the configured Smart Escrow milestone vector is empty. Additionally, updating a Smart Escrow also does not validate (see [\[A04\] Bypass of the Escrow Initialization Validations](#)) the passed Smart Escrow properties. Once these findings have been addressed, and respective milestone emptiness checks added to both Smart Escrow initialization and Smart Escrow update, any other emptiness checks can be removed from this Smart Contract.

There are three instances where the milestones are checked for emptiness:

- in `validate_release_conditions` at `validators/escrow.rs:21-23`
- in `validate_milestone_status_change_conditions` at `validators/milestone.rs:15-17`
- in `validate_milestone_flag_change_conditions` at `validators/milestone.rs:36-38`

Each of these three cases reverts if the milestone list is empty, see e.g.

`validators/escrow.rs:21-23`:

```
if escrow.milestones.is_empty() {
    return Err(ContractError::NoMileStoneDefined);
}
```

Unused admin

The admin address that is set in `EscrowContract::__constructor` in instance storage with key `DataKey::Admin` is unused. Therefore the admin could be removed entirely from the Smart Contract.

Unnecessary use of the function `clone`

The codebase makes use of `.clone()` to pass objects with ownership at function invocations at several places that instead can or could use an immutable borrow instead. This could potentially optimize the efficiency of the contract.

E.g., at `contract.rs:46`, a clone of `e: Env` is passed with ownership, because the invoked function `initialize_escrow` currently declares the parameter `e: Env`. This parameter, and subsequent invoked functions, can be changed to `e: &Env` instead.

Conditional at `core/escrow.rs:17-21`

The function `get_receiver` at `core/escrow.rs:17-21` implements a function to retrieve the `receiver` unnecessarily complicated. The following implementation could be replaced with a simpler one:

```
##[inline]
pub fn get_receiver(escrow: &Escrow) -> Address {
    if escrow.roles.receiver == escrow.roles.service_provider {
        escrow.roles.service_provider.clone()
    } else {
        escrow.roles.receiver.clone()
    }
}
```

Replace the above code with the code below:

```
##[inline]
pub fn get_receiver(escrow: &Escrow) -> Address {
    escrow.roles.receiver.clone()
}
```

Remove `signer` used in `core/escrow.rs:102`

The parameter `signer` in the Smart Escrow public function `get_multiple_escrow_balances` could be removed. It is solely used in `EscrowManager::get_multiple_escrow_balances` at `core/escrow.rs:102` to authenticate the transaction, while the `signer` is not further validated. This use of `require_auth` can be removed.

Disallow `resolve_dispute` with `total_funds = 0`

Currently, the Smart Escrow public function `resolve_dispute` allows for the sum of the parameters `approver_funds + receiver_funds` to be equal to zero. This serves no purpose, as resolving a dispute with a Smart Escrow balance of zero does not transfer any assets.

At the same time, `resolve_dispute` implements additional checks at `calculators.rs:85-91` and `calculators.rs:93-99` to avoid dividing by zero in case the divisor `total_resolved_funds` is zero. This check can be removed by additionally validating that the sum is greater than zero.

Unnecessary checks in `resolve_dispute`

These checks are performed in `validators/dispute.rs:31-37` for `resolve_dispute` that are redundant, as long as [A06] Smart Escrow uses signed integer type `i128` without sign checks is resolved. [A06] Smart Escrow uses signed integer type `i128` without sign checks discusses, among other things, that the fee attributes, including `platform_fee`, might be negative. For this optimization, fees must not be allowed to be negative.

```
if approver_funds < fee_result.net_approver_funds {
    return Err(ContractError::InsufficientApproverFundsForCommissions);
}

if receiver_funds < fee_result.net_receiver_funds {
    return Err(ContractError::InsufficientServiceProviderFundsForCommissions);
}
```

The `net_approver_funds = approver_funds - approver_fees <= approver_funds` always holds, as long as the fees are never negative as previously discussed, due to `calculators.rs:85-91`

```
let net_approver_funds = if total_resolved_funds > 0 {
    let approver_fee_share =
        SafeMath::safe_mul_div(approver_funds, total_fees, total_resolved_funds)?;
    BasicMath::safe_sub(approver_funds, approver_fee_share)?;
} else {
    0
};
```

The same holds for `receiver_funds`, akin to `approver_funds`.

Iterative creation of Soroban vectors instead of patching in-place

Soroban vectors are stored in the host-environment as immutable vectors. Therefore, each mutation of a vector incurs an overhead.

The multi-release smart escrow updates the `flags` of a single milestone by copying previous milestones iteratively into a new empty vector and changing the flags of the respective milestone simultaneously. Considering the aforementioned overhead of Soroban vectors, this is less efficient than changing the respective milestone in-place.

See e.g. `contracts/escrow/src/core/dispute.rs:113` or
`contracts/escrow/src/core/escrow.rs:63`:

```
let mut updated_milestones = Vec::new(&e);
for (index, milestone) in escrow.milestones.iter().enumerate() {
    let mut new_milestone = milestone.clone();
    if index as i128 == milestone_index {
        new_milestone.flags.disputed = true;
    }
    updated_milestones.push_back(new_milestone);
}
```

Consider refactoring to use instead in-place mutation of the existing `milestones` vector.

Unnecessary write to storage

The multi-release smart escrow writes an unchanged `escrow` to storage in the smart contract function `fund_escrow`. Consider removing this redundant write:



```
let escrow = EscrowManager::get_escrow(e.clone())?;
let token_client = TokenClient::new(&e, &escrow.trustline.address);

token_client.transfer(&signer, &e.current_contract_address(), &amount_to_deposit);

e.storage().instance().set(&DataKey::Escrow, &escrow);
```

Unused enumerator iterator and vector length check in for loop

The multi-release smart escrow uses for loops with enumerator iterators without using the provided enumerator index. Additionally, the for loop is wrapped in a redundant conditional that checks whether the vector that is being iterated on is empty. Consider removing the enumerator as well as the redundant vector emptiness check.

See e.g. [contracts/escrow/src/core/validators/escrow.rs:49](#) or [contracts/escrow/src/core/validators/escrow.rs:80](#):

```
if !milestones.is_empty() {
    for (_, milestone) in milestones.iter().enumerate() {
        if milestone.flags.disputed {
            return Err(ContractError::MilestoneOpenedForDisputeResolution);
        }
        if milestone.flags.approved {
            return Err(ContractError::MilestoneApprovedCantChangeEscrowProperties);
        }
    }
}
```

Status

This finding has been addressed in commit IDs `4ee04c2`, `c87c1a8`, `ecbe0be`, `7b9c2bb`, `5107d34`, `7639179`, `af79c87`, `32c985a` of the branch `develop`, and in commit IDs `bba602a`, `9dd0d39`, and `a97eefe` of the branches `multi-release-develop` and `stellar-multi-release-audit`.

[B02] Initialization possible with zero milestones

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

At initialization, an `Escrow` struct is passed that defines the Smart Escrow properties, including the number and details of the milestones. For this purpose, the passed `Escrow` struct is validated with the function `validate_initialize_escrow_conditions`.

This function checks whether a maximum length that is hardcoded to 10 is not exceeded. However, it does not check whether the milestones vector is empty.

A Smart Escrow with no milestones serves no purpose, and the funds cannot be paid out with `release_funds` either, as this function checks whether the milestone vector is empty. As the other function to retrieve funds, `resolve_dispute`, does not check the length of the milestones vector, so funds are not locked in such a Smart Contract. They can still be retrieved with `resolve_dispute`, albeit with fees.

Recommendation

We recommend validating during initialization and Smart Escrow update that the milestone vector is not empty.

Status

This finding has been addressed in commit `ecbe0be` in the branch `develop`.

[B03] At initialization, the `decimals` attribute of `Trustline` is not validated

Severity: Low

Recommended Action: Fix Code

Addressed by client

Description

The Smart Escrow uses the token specified in `Escrow::trustline`, which contains both the token address and its number of decimals.

The attribute `decimals: u32` of type `Trustline` is exclusively used in the public Smart Escrow function `get_multiple_escrow_balances` that returns the balance and decimals of multiple Smart Escrow specified in the function call.

The `decimals` that are returned can hold an arbitrary value specified during initialization of the Smart Escrow that might not match the actual decimals value of the respective token used.

Recommendation

We recommend that the Smart Escrow fetch the `decimals` attribute at runtime instead by calling `TokenClient::decimals`.

Status

The finding has been addressed in the commit ID `c76c7dd` in the branch `develop`.



[B04] `validate_release_conditions` returns wrong error when checking whether funds were already released

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The function `validate_release_conditions` checks whether the funds were already released and in that case returns an error. However, the function incorrectly returns an error indicating that the Smart Escrow was already resolved, rather than specifying that the funds were already released.

See [validators/escrow.rs:13-15](#) :

```
if escrow.flags.released {  
    return Err(ContractError::EscrowAlreadyResolved);  
}
```

Recommendation

We recommend to create a new error in `ContractError` that indicates that the Smart Escrow was already released and use that error instead.

Status

This finding has been addressed in commit IDs `d77d7ae` and `5b9344f` in the branch `develop`.

[B05] `validate_release_conditions` does not check whether the Smart Escrow has already been resolved

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The function `validate_release_conditions` does not check whether the Smart Escrow has already been resolved. Though, as observed in [B04] `validate_release_conditions` returns wrong error when checking whether funds were already released, the error `ContractError::EscrowAlreadyResolved` is falsely returned instead of a new error that properly indicates that the Smart Escrow has already been released.

At the same time, this error indicates that the intent was to also check that the Smart Escrow has not already been resolved yet with `resolve_dispute`.

Recommendation

We recommend adding a missing check in `validate_release_conditions` that the Smart Escrow has already been resolved.

Status

This finding has been addressed in commit ID `5b9344f` in the branch `develop`.

[B06] `dispute_resolver` should consent to Smart Escrow prior to funding the escrow

Severity: Low

Recommended Action: Fix Design

Not addressed by client

Description

Currently, a Smart Escrow can be initialized and used right away with arbitrary role allocations, including the `dispute_resolver` that is typically managed by the platform. This means that the initializing party can choose an arbitrary platform fee and specify the `dispute_resolver` address. In case of a dispute, including scenarios where leftover assets need to be returned, the `dispute_resolver` must act; otherwise, the funds are locked in the Smart Escrow. If the `dispute_resolver` does act, the arbitrarily specified platform fees are enforced.

Recommendation

To avoid such a deadlock situation, the `dispute_resolver` could be required to explicitly consent to a given Smart Escrow after initialization or Smart Escrow updates through `update_escrow`. This can be done with a new Smart Escrow public function

```
EscrowContract::consent(e: Env, dispute_resolver: Address, escrow_properties: Escrow)  
. At the same time, fund_escrow must fail, in case the Smart Escrow has not been consented to by the dispute_resolver.
```

Status

The client has acknowledged this finding.

[B07] Smart Escrows Are Deployers Despite Having Constructors

Severity: Informative

Recommended Action: Fix Code

Not addressed by client

Description

The Smart Escrow, which also has a constructor, has the capabilities of deploying and initializing other contracts through its `deploy` endpoint, making it a deployer. The primary purpose of deployer contracts on Soroban was to serve as a workaround for the initial absence of constructors. They were used to deploy and initialize a contract atomically, mitigating the risk of front-running attacks. Since Soroban now natively supports constructors via the `__constructor` function, a separate deployer contract is no longer necessary to perform this initialization.

The coexistence of both a deployer function and a constructor creates redundancy and is an inefficient design pattern. The current constructor is underutilized, as it only sets an `admin` address in storage that is never referenced in subsequent logic. This design choice is problematic because it introduces complexity without providing any functional benefit. It would be more secure and efficient to remove the deployer logic and use the constructor to handle all necessary initialization tasks. This would streamline the contract's deployment process and reduce its attack surface.

Recommendation

To resolve this issue, we recommend either of the following two solutions. The first is to completely remove the constructor and refactor the deployer function to handle all contract initialization. The second, and more robust, solution is to remove the unnecessary deployer functionality and adapt the existing constructor to perform all essential setup tasks.

Status

The client has acknowledged this finding.

[B08] Disputed Funds Don't Go Back to Funders

Severity: Medium

Difficulty: Low

Recommended Action: Fix Design

Not addressed by client

Description

The Smart Escrow contract, which can be used for crowdsourcing, has a limitation in its dispute resolution mechanism. While the contract facilitates multi-party funding for a Service Provider's project milestones, its logic for handling disputes does not consider the possible multiplicity of funders. When a dispute is triggered and resolved, the funds are returned to the Approver, rather than being refunded to the original funders. This creates an immediate risk for project backers, as they could lose their investment if a project fails and a dispute occurs, unless the Approver can keep track of funders.

Although it may be the case in some scenarios, the Approver is not the source of the funds and, as such, should not be the recipient of a refund. This exposes the funders to a potential loss of capital, as they are entirely reliant on the Approver's integrity and willingness to manually return the funds. The contract's current logic does not guarantee the repayment of funds to the rightful owners, leaving funders dependent on trust in what is intended to be a trustless environment.

Recommendation

To rectify this, the Smart Escrow contract's dispute resolution logic must be updated. The Smart Escrow and platform must either:

1. Store the funders' addresses in persistent data, in pairs with their supplied amounts, implementing an endpoint to allow funders to retrieve their supplied amounts in case of a dispute, or;
2. Keep track of all funders in off-chain storage and find a way to ensure that the approver will repay all funders with their due amounts. This approach, although feasible, requires an extra layer of off-chain trust.

Status

The client has acknowledged this finding.

[B09] The Trustless Work Platform Does Not Handle Storage Archival

Severity: Informative

Recommended Action: Fix Code

Not addressed by client

Description

When a Soroban smart contract is deployed on the Stellar network, its ledger entries, which contain the contract's code and data, are given a Time to Live (TTL). This TTL is a specific number of ledgers, and if it's not extended, the ledger entries will eventually expire and be moved to an archive. This archival process is a core feature of Stellar's state management, designed to keep the network's active ledger size manageable and performant by removing dormant data. When a contract's ledger entries are archived, the contract becomes inaccessible for invocations until restored. This means any funds or tokens held within it become inaccessible until the contract is restored from the archive. This process is a key distinction from traditional blockchains, where a contract's state is perpetually part of the active chain.

To prevent a contract from being archived, developers must actively manage its TTL. This is typically done by submitting transactions that extend the contract's ledger entry TTL. A common practice is to call a function on the contract that performs a state read or write, as any access to a contract's storage automatically resets its TTL. For contracts that are not frequently used, a dedicated "ping" or "keep-alive" function can be called at regular intervals to ensure the contract's storage remains active. If a contract has already been archived, it can be brought back by submitting a special transaction called a restore footprint ([RestoreFootprintOp](#) transaction). This transaction informs the network that the contract's data is needed again, and the Stellar network will bring the latest archived ledger entries back into the active state. This feature ensures that even if a contract expires, its state is not lost forever and can be recovered when needed.

Smart Escrows have no means of updating their own TTL according to the implemented business logic, and no [RestoreFootprintOp](#) is constructed in the platform's backend.

Recommendation

To avoid needing to manually reinstate the Smart Escrow data after properly modifying the storage entries from persistent to instance, we recommend extending the time-to-live of the contract data within the contract itself whenever the contract is interacted with. For example, whenever fetching



information from Smart Escrows using the `get_escrow` endpoint, extend the contract instance storage TTL.

Additionally, the backend should be able to construct `RestoreFootprintOp` operations to bring back contracts from archival.

Potentially helpful resources: [\[1\]](#), [\[2\]](#).

Status

The client has acknowledged this finding.

[B10] The Milestone Updates Phase Can Be Skipped

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

According to [Trustless Work's documentation](#), the expected lifecycle of an escrow is composed of 5 stages:

1. Initiation Phase: The foundation of the process, where roles, responsibilities, and transaction terms are established, and the escrow contract is created.
2. Funding Phase: The phase where the funds are deposited into the escrow contract, securing the transaction and preparing for the next steps.
3. Milestone Updates Phase: As the transaction progresses, milestones are marked as completed by the designated party, providing visibility and enabling reviews.
4. Approval Phase: The phase where milestones are reviewed and approved (or disputed), moving the transaction closer to resolution.
5. Release Phase: Funds are released to the designated recipient based on milestone approvals or dispute resolutions, completing the financial component of the transaction.

Although designed to follow this lifecycle, there are no validations in the code enforcing that the Milestone Updates Phase is necessary. Milestones can be approved regardless of whether their status is updated or not.

Recommendation

Either recognize the Milestone Updates Phase as optional or enforce its necessity in the contract's business logic.

Status

This finding has been addressed in commit ID [93c4c37](#) and [898097a](#) in the `multi-release-develop` and `stellar-multi-release-audit` branches.

[B11] It Is Possible To Change The Platform Address Of A Smart Esrow

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The platform address, as previously discussed in the [Platform Features and Logic Description](#), is the authority representing the platform or entity responsible for the mediation of the agreement between the service provider and, primarily, funders. Due to its role, the platform address is the only address capable of modifying the properties of a deployed smart escrow, subject to a specific set of validations.

The validations implemented when attempting to update a smart escrow do not consider the possibility of the platform address itself being modified. This may introduce trust issues related to the platform entity and strongly suggests that users should manually check the state of a smart escrow prior to performing operations on it.

"This issue is informational because smart escrows cannot be updated after being funded, which narrows the window in which this property could be exploited.

Recommendation

Make the platform address immutable by enforcing immutability at the contract level.

Status

The client has addressed this finding in commit ID [118a316](#) of the branch [single-release-develop](#).

[B12] Differences Between the Single and Multi-Release Escrows

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

While reviewing the code of the Multi-Release Smart Escrow version, multiple minor differences between the implementations have been found in sections that implement the same logic. These differences may hinder the maintainability of both versions of the contracts."

One of these differences, however, is more significant. When resolving a milestone dispute in the Multi-Release version, one of the parameters is named `service_provider_funds`, with its naming indicating that it holds the amount of funds to be sent to the service provider. However, tracing the logic shows that these funds are actually sent to the receiver instead. In the Single-Release version, this variable is appropriately named `release_funds`.

Recommendation

Align the implementations of shared logic across both contract versions to improve maintainability and reduce confusion.

Status

This finding has been addressed in the commit ID `699e25e` of the branches `multi-release-develop` and `stellar-multi-release-audit`.



Backend Findings

This section contains the findings related to the design review performed over the backend code, which orchestrate the interactions between the front-end and, consequently, users, and the Smart Escrows.

[TS01] In-Memory Queue Storage Causing Data Loss and Scaling Issues

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The `PendingWriteQueueService` uses an in-memory `Map<string, PendingWriteItem>()` for transaction queue storage, which introduces multiple critical issues:

- 1. Data Loss on Server Restart/Crash:** All pending transactions are lost when the server restarts, leading to incomplete blockchain operations that cannot be recovered. Users may experience failed transactions with inconsistent state.
- 2. No Horizontal Scaling Support:** Each server instance has an isolated queue, so load balancer routing to different instances breaks transaction flow. A transaction started on Server A will fail if the completion request hits Server B.
- 3. Memory Leak Risk:** Abandoned transactions are never removed from the queue. There's no TTL (Time To Live) mechanism for stale entries, which may cause the queue to grow indefinitely without cleanup.
- 4. No Observability:** Cannot monitor queue size or processing metrics, no visibility into failed transactions across restarts, making it difficult to debug transaction issues.

Recommendation

Replace the in-memory storage with a persistent, distributed solution:

```
// Option 1: Redis-based queue
import { Queue } from "bull";
const pendingWriteQueue = new Queue("pending writes", {
  redis: { host: "redis-server", port: 6379 },
});

// Option 2: External message queue (AWS SQS, RabbitMQ, etc.)
```

Implement proper cleanup mechanisms and monitoring for queue health.



Status

This finding has been addressed in commit IDs [f353f3c](#), [0bf1ac9](#), [98b61ab](#), and [4412028](#).

[TS02] Open CORS Policy Allowing Unauthorized Cross-Origin Requests

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

The `main.ts` file has an overly permissive CORS configuration that allows requests from any origin:

```
app.enableCors({
  origin: true, // ✗ Allows ANY origin
  methods: "GET,HEAD,PUT,PATCH,POST,DELETE",
  credentials: true, // ✗ Dangerous with origin: true
});
```

This configuration creates several security vulnerabilities:

1. **Cross-Site Request Forgery (CSRF)**: Any website can make authenticated requests to the API
2. **Data Theft**: Malicious sites can access user data through browser requests
3. **API Abuse**: Unauthorized third parties can consume API resources without restriction

Recommendation

Restrict CORS to specific, trusted domains:

```
app.enableCors({
  origin: ["https://yourdomain.com", "https://app.yourdomain.com"],
  methods: "GET,HEAD,PUT,PATCH,POST,DELETE",
  credentials: true,
});
```

Use environment variables to manage different origins for different environments (development, staging, production).

Status

This finding has been addressed in commit ID [4a82d73](#).

[TS03] Missing Authentication Guards on Critical Endpoints

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

Multiple critical endpoints lack proper authentication guards, making them publicly accessible:

1. **User Endpoints:** All `/user/*` API endpoints don't require authentication. Users can:

- Get all users data via `GET /user/get-all`, including sensitive `apiKey` information
- Update any user's data as long as they have the user's ID
- Access user profiles without authorization

2. **Notification Endpoints:** `notifications.controller.ts` endpoints lack authentication guards, allowing:

- Unauthorized access to notification data
- Public access to sensitive notification information
- Uncontrolled access to notification operations

This creates severe security vulnerabilities, as sensitive user data and system operations are exposed to unauthorized users.

Recommendation

Add authentication guards to all protected endpoints:

```
@UseGuards(AuthGuard())
@ApiBearerAuth("jwt-auth")
```

Implement role-based access control (RBAC) for different user types and operations. Ensure that users can only access and modify their own data unless they have explicit administrative privileges.

Status

This finding has been addressed in commit IDs `4f7b37e`, `859e8d9`, `fe8a6f6`, and `2e8b7cf`.

[TS04] Authorization Bypass in Escrow Repository

Severity: Medium

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The `escrow.repository.ts` file contains a critical authorization bypass vulnerability in the `findByAdvancedQuery` method. When both `signer` and `contractIds` parameters are provided, the method only validates against `contractIds` but ignores the `signer` validation:

```
} else if (contractIds && signer) {  
    query = this.col.where("contractId", "in", contractIds);  
    // ❌ Missing signer validation - allows access to any contract regardless of signer  
}
```

This allows users to access escrow contracts they shouldn't have access to by providing any valid contract IDs, bypassing the signer-based authorization mechanism.

Security Impact:

- **Authorization Bypass:** Users can access escrows they don't own or have rights to
- **Data Exposure:** Contract data becomes accessible without proper authorization
- **Privilege Escalation:** Users can potentially view or interact with high-value escrows

Recommendation

Add proper signer validation to the query:

```
} else if (contractIds && signer) {  
    query = this.col  
        .where("contractId", "in", contractIds)  
        .where("signer", "==", signer);  
}
```

Implement comprehensive authorization checks throughout the escrow repository to ensure users can only access contracts they are authorized to view or modify.



Status

This finding has been addressed in commit ID [f3ac8da](#).

[TS05] Information Disclosure Through Error Messages

Severity: Low

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The `user.repository.ts` file exposes internal system information through detailed error messages:

```
throw new NotFoundException(`User ${id} not found`);  
throw new ConflictException(  
  `User with address ${data.address} already exists`,  
);
```

These detailed error messages introduce multiple security vulnerabilities:

1. **User Enumeration:** Attackers can systematically discover valid user IDs by analyzing different error responses
2. **System Information Leak:** Internal identifiers and data structures are exposed to potential attackers
3. **Attack Vector Discovery:** Detailed errors help attackers infer the system architecture and identify potential attack vectors

This information can be used by malicious actors to map out the user base and system structure, facilitating more targeted attacks.

Recommendation

Use generic error messages for client responses while logging detailed information separately for debugging:

```
// Generic response to client  
throw new NotFoundException('User not found');  
  
// Detailed logging for debugging (server-side only)  
this.logger.warn(`User lookup failed: User ${id} not found`);
```



Implement a consistent error handling strategy that:

- Returns generic, non-revealing error messages to clients
 - Logs detailed information server-side for debugging
 - Avoid exposing internal system details, including whether a specific user exists.
-

Status

This finding has been addressed in commit IDs [cb5d5f6](#), and [b3f9255](#).

[TS06] HTTP Connections Allowed in Production Environment

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

Multiple locations in the codebase allow HTTP connections to Stellar RPC servers by using `{ allowHttp: true }`. This configuration permits unencrypted communication with blockchain nodes, which poses significant security risks in production environments.

Security Risks:

- Man-in-the-Middle Attacks:** HTTP traffic can be intercepted and potentially modified by attackers
- Data Exposure:** Sensitive blockchain transaction data is transmitted in plain text
- Transaction Tampering:** Unencrypted communication channels can be exploited to manipulate transaction data

This is particularly concerning for a financial application handling escrow transactions, where data integrity and confidentiality are critical.

Recommendation

Configure the application to use HTTPS-only connections in production:

```
// Use environment-specific configuration
const rpcConfig = process.env.NODE_ENV === 'production'
? { /* HTTPS only - remove allowHttp */ }
: { allowHttp: true }; // Only allow HTTP in development

// Or explicitly enforce HTTPS
const rpcConfig = {
  allowHttp: false // Force HTTPS connections
};
```

Implement environment-specific configurations that:

- Force HTTPS connections in production and staging environments



- Allow HTTP only in local development environments
 - Add runtime checks to prevent accidental HTTP usage in production
-

Status

This finding has been addressed in commit ID [6cc12b5](#).

[TS07] Unsafe HTTP Methods for State-Changing Operations

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The `notifications.controller.ts` uses GET HTTP methods for operations that change system state. This violates REST principles and creates security vulnerabilities:

```
// Using GET for state-changing operations (incorrect)
@Get('/endpoint-that-changes-state')
someStateChangingOperation() {
    // Modifies system state
}
```

Security and Design Issues:

- HTTP Semantics Violation:** GET requests should be idempotent and safe (no side effects)
- CSRF Vulnerability:** State changes via GET can be exploited through Cross-Site Request Forgery attacks
- Unintended Side Effects:** GET requests can be cached, prefetched, or triggered by web crawlers, causing unintended state changes
- Security Tool Confusion:** Security scanners and firewalls may not properly analyze GET requests for malicious state changes

CSRF Vulnerability:

Websites can use safe requests such as GET to load static resources for websites such as images, e.g.: ``. This can be exploited to make clients of such websites send malicious GET requests to other websites. Cookies that are marked as `SameSite=None` are always sent alongside these requests. Cookies that are marked as `SameSite=Lax` are only added to safe requests such as GET. In case the cookies are to be utilized by the backend, this could be a source of potential security vulnerabilities, as cookies are typically marked as `SameSite=Lax`.

Recommendation

Change GET methods to appropriate HTTP methods for state-changing operations:

```
// Use POST for operations that create or modify state
@Post('/endpoint-that-changes-state')
someStateChangingOperation() {
    // Modifies system state
}

// Use PUT for idempotent updates
@Put('/resource/:id')
updateResource() {
    // Updates existing resource
}

// Use DELETE for deletion operations
@Delete('/resource/:id')
deleteResource() {
    // Removes resource
}
```

Follow REST conventions:

- GET: Retrieve data (read-only, idempotent)
- POST: Create new resources or non-idempotent operations
- PUT: Update existing resources (idempotent)
- DELETE: Remove resources

Status

This finding has been addressed in commit ID [27c7ebe](#).

[TS08] Type Safety Issues - From 'any' Types to DocumentData Casting

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The codebase previously made extensive use of the `any` type throughout multiple files, which compromised TypeScript's type safety benefits. This originally affected:

Repository Layer:

- `user.repository.ts` : Functions like `getAllUsers`, `findById`, `create`, `addApiKey`, `update`, `updateAddress` returned `Promise<any>`
- `escrow.repository.ts` : Variables like `const raw = doc.data() as any` bypassed type checking
- `escrow-firebase.service.ts` : Parameters like `escrowProperties: any` and return types with `data?: any`

Utility Functions:

- `firebase.utils.ts` : Type annotations like `error: any` in try/catch blocks
- `parse.utils.ts` : Multiple functions used `any` for parameters and return types
- `stellar-transaction-builder.service.ts` : Various `any` types throughout

Progress Made:

The explicit `any` types have been largely removed from the codebase, which is a significant improvement.

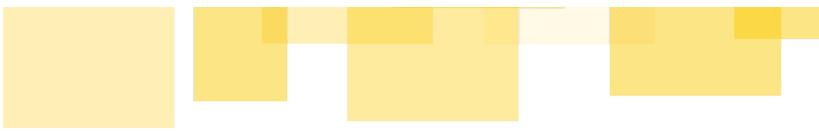
Remaining Issue:

While the code now uses `admin.firestore.DocumentData` (which is better than `any`), it still lacks specific type annotation:

Current Issue:

```
const escrow = doc.data(); // Type: FirebaseFirestore.DocumentData
```

Areas Needing Specific Type Casting:

- 
- `escrow.repository.ts`: Variables like `const raw = snap.data()!` and `const raw = doc.data()` use generic `DocumentData`
 - `notifications.service.ts`: `const escrow = escrowDoc.data()`, `const escrow = doc.data()` in multiple functions lack proper type annotations
 - `user.repository.ts`: Document data retrieval functions return generic `DocumentData` instead of `User` types
 - `escrow-firebase.service.ts`: Firestore operations use generic types instead of specific escrow interfaces

Type Safety Impact:

- Loss of IntelliSense and autocomplete for specific document properties
- No compile-time checking for property access
- Potential runtime errors when accessing properties that may not exist
- Reduced code maintainability and documentation value

Recommendation

Cast `DocumentData` to specific types that match the actual document structure:

```
// Instead of:  
const escrow = doc.data(); // Type: DocumentData  
  
// Use specific casting:  
const escrow = doc.data() as FirestoreEscrowDocument;  
// or  
const escrow = doc.data() as SingleReleaseEscrow;  
  
// For notifications service:  
const escrow = escrowDoc.data() as FirestoreEscrowDocument;  
  
// For user repository:  
const userData = doc.data() as FirestoreUserDocument;  
  
// Create specific Firestore document interfaces:  
interface FirestoreEscrowDocument {  
  contractId: string;  
  title: string;  
  description: string;  
  amount?: number;
```

```
status: string;
createdAt: FirebaseFirestore.Timestamp;
updatedAt: FirebaseFirestore.Timestamp;
// ... other escrow properties
}

interface FirestoreUserDocument {
  address: string;
  email?: string;
  name?: string;
  apiKey?: string;
  createdAt: FirebaseFirestore.Timestamp;
  // ... other user properties
}
```

Benefits of Specific Type Casting:

- Full IntelliSense and autocomplete support
- Compile-time property validation
- Better code documentation
- Reduced runtime errors
- Improved refactoring safety

Status

This finding has been addressed in commit IDs `67c890c`, `bdf8d64`, `4b5a87a`, `f0b34c3`, `55fa23b`, and `01724da`.

[TS09] Incorrect Validation Decorators for Numeric Fields

Severity: Low

Recommended Action: Fix Code

Addressed by client

Description

The `single-release-escrow.class.ts` file contains incorrect validation decorators where numeric fields are decorated with `@IsString()` instead of the appropriate `@IsNumber()` decorator:

```
// ✗ Incorrect validation
@IsString()
amount: number;

@IsString()
approverFunds: number;

@IsString()
receiverFunds: number;

@IsString()
platformFee: number;
```

This mismatch between the validation decorators and the actual data types creates several issues:

- 1. Runtime Validation Failures:** String validation will fail when receiving valid numeric inputs
- 2. Type Safety Compromise:** The type system expects numbers but validation expects strings
- 3. API Contract Confusion:** Unclear whether the API expects strings or numbers
- 4. Potential Data Corruption:** Inconsistent type handling can lead to incorrect data processing

Recommendation

Use the correct validation decorators that match the TypeScript types:

```
// ✓ Correct validation
@IsNumber()
amount: number;

@IsNumber()
```

```
approverFunds: number;
```

```
@IsNumber()
```

```
receiverFunds: number;
```

```
@IsNumber()
```

```
platformFee: number;
```

Additionally, consider adding more specific numeric validations:

```
@IsNumber()
```

```
@IsPositive() // Ensure positive values for amounts
```

```
@Min(0.01) // Minimum transaction amount
```

```
amount: number;
```

Review all DTO classes to ensure validation decorators match their corresponding TypeScript types.

Status

This finding has been addressed in commit ID [ee82a6e](#).

[TS10] Client-Side Timestamps Creating Data Inconsistency

Severity: Low

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

Multiple files throughout the codebase use client-side timestamps (`new Date()`) instead of server-side timestamps for Firestore operations. This creates several critical issues:

Problems with Client-Side Timestamps:

1. **Timezone Issues:** Clients in different timezones generate inconsistent timestamps
2. **Clock Drift:** Client system clocks may be inaccurate or out of sync
3. **Non-Atomic Operations:** Client timestamps don't reflect the actual database write time
4. **Security Risk:** Clients can manipulate timestamps maliciously

Affected Files:

- `repositories/escrow.repository.ts` : Uses `new Date()` for `updatedAt` and `createdAt`
- `stellar-contract/escrow/firestore-services/escrow-firebase.service.ts` : Uses `new Date()` for `updatedAt`
- `stellar-contract/queue/pending-write-handler.service.ts` : Uses `new Date()` for `updatedAt`
- `firebase/converters/escrow.repository.ts` : Uses `new Date()` as fallback values

Current Problem:

```
// Inconsistent client-side timestamps
isActive: true,
updatedAt: new Date(),
createdAt: new Date(),
```

Recommendation

Replace client-side timestamps with `FieldValue.serverTimestamp()`:



```
import { FieldValue } from 'firebase-admin/firestore';

// Use server timestamps for consistency
isActive: true,
updatedAt: FieldValue.serverTimestamp(),
createdAt: FieldValue.serverTimestamp(),
```

Benefits of server timestamps:

- Consistent timezone (server timezone)
- Accurate timing (actual database write time)
- Atomic operations (timestamp is set when the write occurs)
- Security (clients cannot manipulate timestamps)
- Proper ordering of operations

Also update milestone timestamps (`approvedAt` , `completedAt`) for consistency.

Status

This finding has been addressed in commit IDs `7942bd4` , and `8360e9b` .

[TS11] Missing Input Validation and Type Annotations

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

Several functions throughout the codebase lack proper input validation and explicit type annotations, creating potential runtime errors and security vulnerabilities:

Missing Type Annotations:

1. `parse.utils.ts` : The `adjustPricesToMicroUSDC` function has missing type annotation for the `decimals` parameter:

```
export function adjustPricesToMicroUSDC(price: number, decimals): string {
```

2. `firebase/convertisers/escrow.repository.ts` : Converter functions lack explicit parameter types:

```
toFirestore: (data) => ({
fromFirestore: (snap: QueryDocumentSnapshot) =>
```

Missing Input Validation:

Functions accept parameters without validating them:

- Finite number checks
- Range validation
- Non-negative number validation
- Null/undefined checks

Security and Reliability Risks:

1. **Runtime Errors:** Invalid inputs can cause unexpected crashes
2. **Data Corruption:** Unvalidated inputs can corrupt calculations
3. **Security Vulnerabilities:** Malicious inputs might exploit undefined behavior

Recommendation

Add explicit type annotations and comprehensive input validation:

```
// Add proper typing and validation
export function adjustPricesToMicroUSDC(price: number, decimals: number): string {
  if (!Number.isFinite(price) || price < 0) {
    throw new Error('Price must be a finite, non-negative number');
  }
  if (!Number.isFinite(decimals) || decimals <= 0) {
    throw new Error('Decimals must be a positive, finite number');
  }
  // ... rest of function
}

// Fix converter type annotations
toFirestore: (data: FirestoreSingleReleaseEscrow) => ({
fromFirestore: (snap: QueryDocumentSnapshot) =>
```

Implement consistent validation patterns:

- Type guards for complex objects
- Range validation for numeric inputs
- Format validation for strings
- Null/undefined checks where appropriate

Status

This finding has been addressed in commit ID [6667272](#).

[TS12] Unreliable Type Guard Functions and Interface Design Issues

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

Unreliable Type Guards

The `isRawSingleEscrow()` and `isRawMultiEscrow()` functions use property presence (`"amount" in data`) as discriminants instead of the explicit `type` field. This approach is fragile and may lead to incorrect type narrowing:

```
// Current fragile approach
export function isRawSingleEscrow(data: RawEscrowData): data is RawSingleEscrow {
  return "amount" in data; // ❌ Fragile - relies on property presence
}
```

Interface Inconsistencies:

Different flag naming conventions between single and multi-release escrows create confusion:

```
interface RawSingleFlags {
  dispute: boolean;    // ❌ Different naming
  release: boolean;   // ❌ Different naming
  resolved: boolean;
}

interface RawMultiFlags {
  disputed: boolean;  // ❌ Different naming
  released: boolean; // ❌ Different naming
  resolved: boolean;
  approved: boolean;
}
```

Missing Interface Consolidation:

`RawSingleEscrow` and `RawMultiEscrow` share common properties but don't extend a base interface, leading to code duplication.



Recommendation

Fix Type Guards to Use Discriminant Field:

```
export function isRawSingleEscrow(data: RawEscrowData): data is RawSingleEscrow {
  return data.type === "single-release";
}

export function isRawMultiEscrow(data: RawEscrowData): data is RawMultiEscrow {
  return data.type === "multi-release";
}
```

Standardize Flag Naming:

Choose consistent naming convention (either past tense or present tense) for all flag interfaces.

Create Base Interface:

```
export interface RawEscrow {
  title: string;
  roles: RawRoles;
  description: string;
  engagement_id: string;
  platform_fee: string | number | bigint;
  trustline: RawTrustline;
  receiver_memo: string | number | bigint;
}

export interface RawSingleEscrow extends RawEscrow {
  type: "single-release";
  amount: string | number | bigint;
  flags: RawSingleFlags;
  milestones: RawSingleMilestone[];
}
```

Status

This finding has been addressed in commit ID [3c84811](#).

[TS13] Environment Configuration Management Issues

Severity: Informative

Recommended Action: Document Prominently

Addressed by client

Description

Environment Variable Inconsistencies:

The `.env.example` file contains inconsistencies with actual code usage:

1. Firebase Configuration Mismatch:

- `.env.example` uses: `FIREBASE_PROD_PROJECT_ID`, `FIREBASE_PROD_CLIENT_EMAIL`,
`FIREBASE_PROD_PRIVATE_KEY`
- Actual code uses: `FIREBASE_PROJECT_ID`, `FIREBASE_CLIENT_MAIL`,
`FIREBASE_PRIVATE_KEY`

2. Unused Environment Variables:

- `ISSUER_ADDRESS`, `USDC_SOROBAN_CIRCLE_TOKEN_TEST`, `API_SECRET_KEY_WALLET`,
`API_PUBLIC_KEY_WALLET` are defined but not used

3. Environment Value Mismatch:

- `.env.example` defines `ENVIRONMENT=LOCAL` but code only uses `DEV` and `PROD`

Configuration Management Issues:

1. Direct Environment Variable Usage:

Code directly accesses `process.env` instead of using NestJS `@nestjs/config` for type safety

Recommendation

Fix Environment Configuration:

```
// Update .env.example to match actual usage
FIREBASE_PROJECT_ID=
FIREBASE_CLIENT_MAIL=
FIREBASE_PRIVATE_KEY=
ENVIRONMENT=DEV
```



Use Typed Configuration:

```
// Instead of process.env.FIREBASE_PROJECT_ID
constructor(private configService: ConfigService) {}
const projectId = this.configService.get<string>('FIREBASE_PROJECT_ID');
```

Clean Up Unused Environment Variables:

Remove unused variables from `.env.example` or implement their usage in the codebase if they are intended for future features.

Status

This finding has been addressed in commit ID `702c7ed`.

[TS14] Code Quality and Documentation Issues

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

Missing API Documentation:

Multiple DTOs lack proper Swagger documentation, making the API difficult to understand and use:

- `LoginUserDto`, `FundWithMemoDto`, `UpdateUserDto` missing `@ApiProperty` decorators
- `notifications/test` controller lacks `@ApiTag("Notifications")`
- Internal endpoints not excluded from Swagger docs

Missing Field Validation:

DTOs lack proper validation decorators for critical fields:

- `CreateUserDto.address` should use `@IsAddressValid()` validator for Stellar address format validation
- `LoginUserDto.wallet` field lacks wallet address validation
- Email fields should use `@IsEmail()` for proper email format validation

Unused Code and Files:

- `src/repositories/index.ts` - empty file
- `src/firebase/convertisers/index.ts` - empty file
- `src/common/dto/escrow-payload.dto.ts` - `EscrowPayloadDto` interface unused
- `exceptions/engagement-id.exception.ts` - `EngagementIdException` never used
- Entire `auth` directory may be obsolete

Inconsistent Naming Conventions:

- Folder naming: `src/stellar-contract/deployer/Dto` vs `dto` elsewhere
- Variable naming: `milestone_index` should be `milestoneIndex` (camelCase)
- Collection names hardcoded throughout instead of using enums

Logging and Error Handling:

- `console.log` used instead of NestJS Logger throughout the codebase

- Inconsistent error handling patterns
- Missing structured logging for debugging and monitoring

Code Structure Issues:

- Redundant code: `.then((escrow) => escrow)` in `helper.service.ts`
- Type coercion: Using `==` instead of `===` in comparisons
- Repository bypass: `EscrowFirestoreService.saveEscrow()` bypasses repository pattern

Duplicated code:

- `helper.service.ts:631:handlePendingWrite` and `indexer.service.ts:23:updateFromTxHash` implement the same functionality
 - `handlePendingWrite` takes two arguments instead of one, but both arguments are expected to always be equal, see only usage at `src/stellar-contract/helper/helper.service.ts:97`

Wrong logging and error messages:

The codebase contains wrong logging and error messages, most likely due to copy and paste of program code:

- `src/utils/firebase.utils.ts:202:createNotification : Error fetching trustline`

Validation pipe handling:

- `src/main.ts:17` defines a global `ValidationPipe` to be used for all endpoints, while some endpoints define an additional similar `ValidationPipe`, such as e.g. the endpoints in `src/users/user.controller.ts`:

Explicit configuration values:

- `@UseGuards(AuthGuard())`: This should reference the respective authentication guard that is supposed to be used explicitly, in case additional authentication guards are added to the backend
 - e.g.: `@UseGuards(AuthGuard('jwt'))` or `@UseGuards(AuthGuard('local'))`

```
// src/main.ts:17
app.useGlobalPipes(
  new ValidationPipe({
    transform: true,
```

```

        whitelist: true,
        forbidNonWhitelisted: true,
    }),
    new ValidationPipe({
        exceptionFactory: (errors: ValidationError[]) => {
            const formattedErrors = errors.reduce(
                (acc, err) => {
                    acc[err.property] = Object.values(err.constraints);
                    return acc;
                },
                {} as Record<string, string[]>,
            );
            return new BadRequestException({
                statusCode: 400,
                error: "Bad Request",
                message: "Validation failed",
                details: formattedErrors,
            });
        },
    )),
);

// src/users/user.controller.ts:25
@Post("create")
@HttpCode(HttpStatusCode.CREATED)
@UsePipes(new ValidationPipe({ whitelist: true, forbidNonWhitelisted: true }))
async createUser(@Body() createUserDto: CreateUserDto) {
    return this.userService.create(createUserDto);
}

```

Inconsistent database schema:

- `src/stellar-contract/queue/pending-write-handler.service.ts:42` writes `approvedAt` for single-release milestones, but not for multi-release milestones

Improper variable names:

`src/stellar-contract/queue/pending-write-handler.service.ts:100` defines the variable `completedAt` that should instead be called `lastUpdatedAt` or similar

- this variable is updated everytime the status is updated instead of only when the status is `complete`



Recommendation

Improve API Documentation:

```
@ApiProperty({
  description: 'User wallet address',
  example: 'GXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
})
wallet: string;

@ApiTag("Notifications")
@Controller('notifications')
```

Add Proper Field Validation:

```
// Use existing custom validators for address validation
@IsAddressValid()
@IsNotEmpty()
address: string;

// Add email validation
@IsOptional()
@IsEmail()
email?: string;

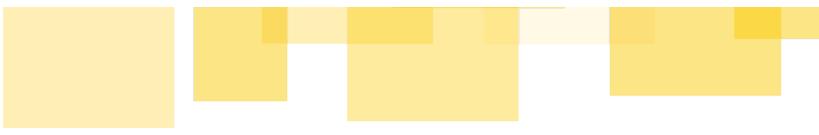
// Apply wallet validation consistently
@IsAddressValid()
@IsNotEmpty()
wallet: string;
```

Clean Up Unused Code:

- Remove empty files and unused interfaces
- Consolidate or remove obsolete `auth` directory
- Remove redundant code patterns
- Reduce duplicated code

Standardize Naming:

```
// Use enums for collection names
enum Collections {
  ESCROWS = 'escrows',
```



```
    USERS = 'users',
    CONTRACT_TEMPLATES = 'contract-templates'
}

// Use camelCase for variables
milestoneIndex
// instead of
milestone_index
```

Implement Structured Logging:

```
// Replace console.log with NestJS Logger
private readonly logger = new Logger(ServiceName.name);
this.logger.log('Operation completed successfully');
```

Use Strict Equality:

```
// Replace == with ===
if (tag === "u64")
// instead of
if (tag == "u64")
```

Duplicated code:

Review the codebase for duplication and consolidate repeated logic where appropriate.

Wrong logging and error messages:

Review all logging and error messages and fix wrong messages accordingly.

Validation pipe handling:

Standardize the validation pipe that is used everywhere and remove redundant local validation pipes.

Explicit configuration values:

Use explicit configuration values, where implicit ones are obscure or might change. E.g., change all occurrences of `@UseGuards(AuthGuard())` with `@UseGuards(AuthGuard('jwt'))`, where JWT authorization is the Auth Guard that should be used.

Inconsistent database schema:

Create a proper schema of the database and review whether the schema is consistent for similar entities such as single-release escrows and multi-release escrow. Ensure that the codebase does not violate the database schema.



Improper variable names:

Review whether all variable names in the codebase accurately reflect the content of the variable or database field.

Status

This finding has been addressed in commit ID [c8e5a65](#).

[TS15] Data Architecture and Repository Pattern Issues

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

Repository Pattern Bypass:

The `EscrowFirestoreService.saveEscrow()` method bypasses the repository layer and calls `addEscrow()` directly, ignoring carefully designed type conversions:

```
// Current: Bypasses repository pattern
await this.firestoreService.addEscrow(docId, escrowProperties);

// Should use: Repository methods for proper type conversion
if (escrowProperties.type === "single-release") {
  await this.escrowRepository.saveSingle(escrowProperties as SingleReleaseEscrow);
}
```

This bypass means that `escrowRepo.saveSingle()` and `mapToFirestoreSingle()` methods are **never called** during escrow deployment, causing data inconsistency and ignoring proper type conversions.

All database queries should be defined in `FirebaseService`

The Firebase service provides the method `getFirestore()` that returns the Firestore instance that can be used to query the database. This leads to query definitions throughout the codebase, instead of having all queries collectively declared in the `FirebaseService`.

Hardcoded Collection Names:

Collection names are hardcoded throughout the codebase instead of using centralized constants:

- `notifications.service.ts : "escrows"`
- `escrow.repository.ts : "escrows"`
- `user.repository.ts : "users"`
- `contract-templates.service.ts : "contract-templates"`

Hardcoded values:

Numerical values are hardcoded as values in the code instead of using constant variables:

- `src/utils/parse.utils.ts:25`: `BigInt(Math.round(price * 1e7))`

Type Safety Issues in Firestore:

- Using `admin.firebaseio.DocumentData` instead of specific types
- `const escrow = doc.data()` lacks proper type annotation
- Loose type definitions that don't match actual usage patterns

Interface Design Questions:

Whether Firestore interfaces should use flat structures (`Record<string, string>`) for easier queries or typed objects (`Roles`, `Trustline`) for better type safety.

NestJS modules:

- `src/config/firebase.config.ts` is not a module, but should be due to providing functions and usage of services

Recommendation

Fix Repository Pattern:

```
// Update EscrowFirestoreService.saveEscrow() to use repository
// Note: Consider adding validation before casting for better type safety
if (escrowProperties.type === "single-release") {
  await this.escrowRepository.saveSingle(
    escrowProperties as SingleReleaseEscrow,
  );
} else {
  await this.escrowRepository.saveMulti(
    escrowProperties as MultiReleaseEscrow,
  );
}
```

Define all database queries in `FirebaseService`

Move all database queries to `FirebaseService` and remove the method `getFirestore` to disincentivize creating queries outside of `FirebaseService`.

Centralize Collection Names:

```
enum Collections {
  ESCROWS = 'escrows',
```



```
    USERS = 'users',
    CONTRACT_TEMPLATES = 'contract-templates',
    NOTIFICATIONS = 'notifications'
}

// Usage
this.col = this.firebaseio.collection(Collections.ESCROWS);
```

Centralize constant magic values:

Create proper constant variables with names and potentially documentation that describe the respective value instead of using undocumented magic values directly in program code.

Improve Type Safety:

```
// Use specific types instead of DocumentData
const escrow = doc.data() as FirestoreEscrowDocument;
```

Standardize Interface Design:

Keep current flat structure for Firestore queries but improve conversion logic between flat and typed structures.

Usage of NestJS modules

Use NestJS modules and services when proper.

Status

This finding has been addressed in commit IDs `d005371`, `c8e5a65`, `a4f8570`, `9dd6787`, and `5cece24`.

[TS16] Development Tooling and Code Quality Setup

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The codebase previously lacked proper development tooling and code quality enforcement mechanisms:

Original Issues:

1. **Missing ESLint Configuration:** No ESLint setup for TypeScript code quality enforcement
2. **No Pre-commit Hooks:** No automated code quality checks before commits
3. **Outdated Husky Version:** Need for modern pre-commit hook management
4. **Missing Linting Integration:** No integration between ESLint and development workflow

Impact of Missing Tooling:

- Inconsistent code formatting and style
- Type safety issues not caught during development
- Manual code review burden increased
- Potential for committing problematic code

Recommendation

Implement comprehensive development tooling, including:

- ESLint with TypeScript support
- Husky for pre-commit hooks
- Prettier for code formatting
- Lint-staged for efficient pre-commit processing

Required Setup:

```
### Install ESLint and related packages
npm install --save-dev @typescript-eslint/parser @typescript-eslint/eslint-plugin eslint

### Install Husky for git hooks
```

```
npm install --save-dev husky  
  
### Install Prettier and integration  
npm install --save-dev prettier eslint-config-prettier eslint-plugin-prettier  
  
### Install lint-staged for efficient pre-commit processing  
npm install --save-dev lint-staged
```

Status

This finding has been addressed in commit IDs [67c890c](#) , [fc3eb3a](#) , [15a3f13](#) , [3a27c6a](#) , [5329a3b](#) , [35c0cba](#) , and [6c354fe](#) .

[TS17] Backend allows for submitting any pre-signed transaction to the Stellar blockchain

Severity: Low

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The function `sendTransaction` at `src/stellar-contract/helper/helper.service.ts:74` takes a signed transaction and sends the transaction to the Stellar blockchain. However, the intended flow is that only transactions originally generated by the backend, signed by the end-user on the client side, and then returned to the backend should be submitted.

However, `sendTransaction` submits a transaction before verifying whether the respective transaction has been created by the backend, i.e., checked whether the transaction hash is in the `pendingWriteQueue`. As a result, the backend can be abused to send arbitrary transactions to the blockchain.

```
const response = await this.horizonServer.submitTransaction(transaction);

const pending = this.pendingWriteQueue.get(txHash);
if (!pending) {
  throw new HttpException(
    {
      status: HttpStatus.BAD_REQUEST,
      message: `No pending write for tx ${txHash}`,
    },
    HttpStatus.BAD_REQUEST,
  );
}
```

Recommendation

Check whether the transaction hash is in the `pendingWriteQueue` before submitting the transaction to the blockchain. In case the hash is not in the queue, abort and do not submit the transaction.



Status

This finding has been addressed in commit ID [9a4006e](#).

[TS18] Singleton State Sharing in Transaction Builder

Severity: Medium

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

The `StellarTransactionBuilderService` is registered as a singleton in NestJS but maintains mutable instance state, resulting in critical race conditions when multiple requests access the service at the same time. This design flaw can lead to transactions being built with incorrect parameters, potentially causing financial losses in the escrow system.

The service is registered as a singleton in `stellar.module.ts` but maintains shared mutable state across all requests:

```
@Injectable()
export class StellarTransactionBuilderService {
    private signer!: string; // X Shared across all requests - User identity
    private contractId!: string; // X Shared across all requests - Contract address
    private wasmHash?: string; // X Shared across all requests - WASM deployment hash
    private method!: string; // X Shared across all requests - Smart contract method
    private operationFunc?: string; // X Shared across all requests - Operation function
    private params: xrdr.ScVal[] = []; // X Shared across all requests - Method parameters
    private constructorArgs?: xrdr.ScVal[]; // X Shared across all requests - Constructor args
    private fee: string = "1000"; // X Shared across all requests - Transaction fee
    private useHorizon: boolean = false; // X Shared across all requests - Server selection
    private addMemo?: string; // X Shared across all requests - Transaction memo
    private queueAction?: { ... }; // X Shared across all requests - Queue operations
}
```

Critical race condition scenarios include:

Scenario 1: Fee Contamination

```
// Request A (SharedEscrowService)
txBuilder.setFee("100").buildUnsignedXDR();

// Request B (SingleReleaseService) - concurrent
txBuilder.setFee("5000").buildUnsignedXDR();
```



```
// ✖ Request A might use fee "5000" instead of intended "100"
```

Scenario 2: Contract ID Cross-contamination

```
// Request A sets contractId "CONTRACT_ABC"
await txBuilder.setContractId("CONTRACT_ABC")...

// Request B sets contractId "CONTRACT_XYZ"
await txBuilder.setContractId("CONTRACT_XYZ")...

// ✖ Request A continues and might use "CONTRACT_XYZ"
```

Scenario 3: Queue Action Mix-up

```
// Request A: Fund escrow operation
txBuilder.addToQueue("FUND_ESCROW", "single-release", {...}, queue);

// Request B: Release funds operation
txBuilder.addToQueue("RELEASE_FUNDS", "multi-release", {...}, queue);

// ✖ Request A might trigger "RELEASE_FUNDS" action instead
```

Many services (`SharedEscrowService`, `SingleReleaseService`, `MultiReleaseService`, and `DeployerService`) use the same singleton instance, creating potential for parameter contamination between concurrent operations across all transaction-related functionality in the system. The `reset()` method in `finally` blocks provides insufficient protection as race conditions occur between method calls, not after completion.

Critical Issue: Incomplete Reset Function

The `reset()` method fails to reset ALL shared fields, making some race conditions persist even longer:

```
reset(): void {
  this.signer = "";
  this.contractId = "";
  this.method = "";
  this.params = [];
  this.useHorizon = false;
  this.addMemo = undefined;
```

```
this.queueAction = undefined;
this.constructorArgs = [];

// ❌ MISSING - These fields are NEVER reset:
// this.fee = "1000";           // Fee persists across requests!
// this.wasmHash = undefined;   // WASM hash persists!
// this.operationFunc = undefined; // Operation func persists!
}
```

This means `fee`, `wasmHash`, and `operationFunc` contamination persists across multiple requests until explicitly overwritten.

This affects all transaction-related operations in the system and can lead to wrong fees, operations on incorrect contracts, or funds released to wrong parties.

Recommendation

We recommend implementing either a request-scoped service or a stateless factory pattern to eliminate shared mutable state:

Option A: Request-Scoped Service

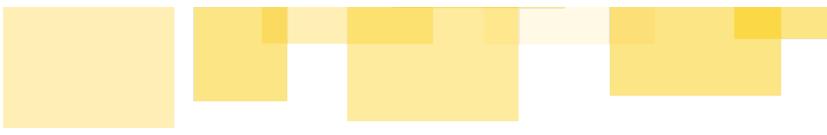
```
@Injectable({ scope: Scope.REQUEST })
export class StellarTransactionBuilderService {
    // Each request gets its own instance
}
```

Option B: Stateless Factory Pattern (Preferred)

```
@Injectable()
export class StellarTransactionBuilderFactory {
    createBuilder(): StellarTransactionBuilder {
        return new StellarTransactionBuilder(this.sorobanServer, this.horizonServer);
    }
}
```

This ensures each operation gets a fresh, isolated builder instance with no shared state between concurrent requests.

Current Problematic Usage:



```
// ❌ Both services share the same mutable instance
// ServiceA
await this.txBuilder.setFee("100").buildUnsignedXDR();

// ServiceB (concurrent)
await this.txBuilder.setFee("200").buildUnsignedXDR();
// ServiceA might get fee "200"!
```

Fixed Usage with Factory:

```
// ✅ Each operation gets fresh, isolated builder
// ServiceA
const builderA = this.txBuilderFactory.createBuilder();
await builderA.setFee("100").buildUnsignedXDR();

// ServiceB (concurrent)
const builderB = this.txBuilderFactory.createBuilder();
await builderB.setFee("200").buildUnsignedXDR();
// No interference between operations
```

Status

This finding has been addressed in commit ID [575a047](#).

[TS19] Insecure private key storage in singleton service instance variables

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The `HelperService` stores sensitive private key material in an instance variable (`sourceKeypair`), which creates security risks in a singleton service architecture. This pattern retains private keys in memory longer than necessary and potentially exposes them across multiple method calls within the same service instance.

```
@Injectable()
export class HelperService {
    private sourceKeypair: StellarSDK.Keypair; // ✗ Private key stored in instance variable

    async establishTrustline(sourceSecretKey: string): Promise<ApiResponse> {
        try {
            this.sourceKeypair = StellarSDK.Keypair.fromSecret(sourceSecretKey); // ✗ Stored in
instance
            const account = await this.sorobanServer.getAccount(
                this.sourceKeypair.publicKey(), // ✗ Used from instance variable
            );
            // ... rest of method uses this.sourceKeypair
            const result = await signAndSendTransaction(
                transaction,
                this.sourceKeypair, // ✗ Private key passed from instance variable
                this.sorobanServer,
                false,
            );
        }
    }
}
```

Security Concerns:

1. **Extended Memory Exposure:** Private keys remain in memory as instance variables until the service is garbage collected or the key is overwritten

- 
2. **Cross-Method Accessibility:** Any method in the service can access `this.sourceKeypair` even if not intended
 3. **Debugging/Logging Risk:** Instance variables are more likely to be accidentally exposed in debug output or error logs
 4. **Memory Dumps:** Private keys stored in instance variables are more likely to appear in memory dumps during debugging

Current Risk Level: While the current usage in `establishTrustline()` method appears to use the key only within the same method scope, storing sensitive cryptographic material in instance variables is an anti-pattern that violates the principle of least exposure.

Recommendation

Limit private key scope to local variables within methods and clear references as soon as they are no longer required:

```
async establishTrustline(sourceSecretKey: string): Promise<ApiResponse> {
  try {
    // ✓ Use local variable instead of instance variable
    const sourceKeypair = StellarSDK.Keypair.fromSecret(sourceSecretKey);

    const account = await this.sorobanServer.getAccount(
      sourceKeypair.publicKey(),
    );

    const usdcAsset = new StellarSDK.Asset("USDC", this.usdcTokenPublic);
    const operations = [
      StellarSDK.Operation.changeTrust({ asset: usdcAsset }),
    ];
    const transaction = buildTransaction(account, operations);

    const result = await signAndSendTransaction(
      transaction,
      sourceKeypair, // ✓ Pass local variable
      this.sorobanServer,
      false,
    );

    // ✓ Optional: Explicitly clear sensitive data
    // sourceKeypair = null; // Clear reference when done
  }
}
```

```
return {
  status: result.status,
  message: "The trust line has been correctly defined in the USDC token",
};
} catch (error) {
  // Error handling...
}
}
```

Additional Security Improvements:

- 1. Remove instance variable:** Delete `private sourceKeypair: StellarSDK.Keypair;` from the class
- 2. Minimize scope:** Use local variables for all cryptographic material
- 3. Clear sensitive data:** Explicitly null sensitive variables when operations complete
- 4. Audit all services:** Review other services for similar private key storage patterns

Status

This finding has been addressed in commit ID [939242c](#).

[TS20] Inefficient database querying

Severity: Low

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

The codebase uses inefficient database queries and filters manually after fetching multiple or all documents of a collection. For example, the following code from

`src/notifications/notifications.service.ts:57` applies a Firestore query and then performs additional manual filtering afterwards.

```
const snapshot = await firestore
  .collection("escrows")
  .where("disputeFlag", "==", true)
  .get();

const HIGH_VALUE_THRESHOLD = 500; // ! ask: how many USDC?
for (const doc of snapshot.docs) {
  const escrow = doc.data() as SingleReleaseEscrow;
  if (escrow.amount > HIGH_VALUE_THRESHOLD) {
    // ! ask: amount or balance?
    await createNotification(this.firebaseioService, {
      contractId: escrow.contractId,
      type: "high_value_dispute",
      title: "High Value Escrow in Dispute",
      message: `Escrow ${escrow.title} with value ${escrow.amount} is in dispute`,
      entities: [
        escrow.roles.approver,
        escrow.roles.serviceProvider,
        escrow.roles.platformAddress,
        escrow.roles.releaseSigner,
        escrow.roles.disputeResolver,
        escrow.roles.receiver,
      ],
    });
  }
}
```

Recommendation



We recommend to adjust all queries to include all relevant filters beforehand, as Firestore supports expressive queries.

Status

This finding has been addressed in commit [5007e8d](#)

[TS21] Notifications are created every hour once they start getting generated

Severity: Low

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

Cron jobs:

The backend uses the `@Cron` annotations from `@nestjs/common` to create functions that are run periodically to create notifications for users. These functions check for criteria such as smart escrow and milestone staleness to remind the users for pending work items.

For example, the function

`src/notifications/notifications.service.ts:17:checkPendingEscrows` creates notifications for smart escrows that have unapproved milestones and are older than a fixed offset (for instance, 7 days). This check is executed every hour. By design, this function will continuously generate notifications until the criteria are not met anymore, thereby generating the same notifications every hour, unnecessarily filling the notification inbox and database with data.

This issue also applies to the other cron jobs declared in

`src/notifications/notifications.service.ts`.

Notification endpoints:

The backend endpoints `notifications/test/*` allow for triggering the notification checks manually for all users. This allows for malicious API users (without any credentials) to spam the endpoint with these notification requests. This will cause the backend to possibly generate multiple notifications per request that are all stored in the database. These endpoints are defined in the file `src/notifications/notifications.controller.ts`.

Recommendation

Cron jobs:

Update a timestamp per notification type that is used to prevent the same notifications from being generated more than once in a given period of time.

Notification endpoints:

Remove the notification endpoints.



Status

This finding has been addressed in commit ID [0652a79](#)



[TS22] No notifications are being created for multi-release escrows

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The methods in `src/notifications/notifications.service.ts` that generate user notifications only consider single release escrows instead of supporting both single-release and multi-release escrows:

- `checkHighValueDisputes`
- `checkInactiveEscrows`
- `checkPendingEscrows`
- `checkCompletedMilestones`

Recommendation

Update these methods to also generate notifications for multi-release escrows for consistent behaviour.

Status

This finding has been addressed in commit ID `ee38c9e`

[TS23] Use of floating-point `number` type for critical values

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The codebase makes extensive use of the JavaScript primitive type `number`, which is a double-precision 64-bit floating point number. Due to the floating-point implementation, the `number` type is prone to rounding errors by nature. Simply parsing a numerical string to a `number` may round the number upwards or downwards, typically for very large or very small numbers.

Due to this, other types and classes should be used to handle critical numbers. For example, the JavaScript primitive type `bigint`, typically called a `BigInt` value/object, can be used to precisely store integers. As JavaScript offers no equivalent for non-integer arbitrary-precision numbers, a library must be used that implements similar functionality for such cases.

Dangerous usage of `number`

The codebase dangerously uses `number` to store and calculate critical values such as e.g.:

- `src/stellar-contract/escrow/Dto/fund-escrow.dto.ts:17` : monetary value `amount`
- `src/utils/parse.utils.ts:21` : currency conversion using imprecise floating-point arithmetic

Recommendation

Review the codebase in regard to usage of `number` and replace `number` with a precise equivalent such as e.g. `BigInt`. Ensure that all arithmetic operations are precise and round towards the correct direction in case rounding is unavoidable. Ensure that serialization and deserialization of numerical values incurs no critical loss of precision.

Status

This finding has been addressed in commit ID `ee82a6e`.

[TS24] API key does not meet security standards

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The backend uses JWT for user session authorization and additionally supports API keys. However, the API keys are currently not used anywhere in the codebase for actual authorization, which makes them serve no purpose. All endpoints that require authorization currently make use of the JWT token.

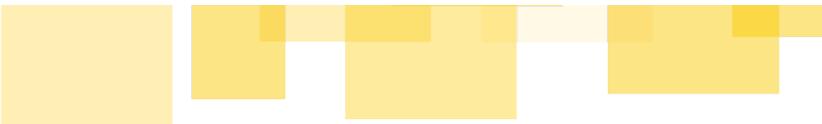
The API key can be added to a user account by using the `user/:id/api-key` endpoint and supplying the key in the body. Currently, this endpoint is not protected with proper authentication and authorization, as described in [\[TS03\] Missing Authentication Guards on Critical Endpoints](#). There also exist other endpoints that similarly allow for adding API keys. Removal of API keys is supposed to happen by supplying an empty array, as described in the comment at `src/repositories/user.repository.ts:140`.

The API keys are saved verbatim in the database. This means that API keys can be retrieved in plain text if an attacker gains read access to the database.

There is no format enforced that the API keys must comply with, other than sometimes enforcing them to be strings. The endpoint `auth/request-api-key` overrides all API keys of the given user with a single API key that is identical to the JWT token used for user session authorization. This endpoint is also not protected, currently allowing anyone to override the configured API keys. Additionally, this unsecured endpoint returns the user data including the generated API key.

Violated security standards for API keys

- Verbatim storage of API keys in the database:
 - API keys are expected to be stored in an in-retrievable way by using state-of-the-art cryptographic methods
- Using the user authorization session JWT as an API key
 - While JWT could be used as an API key, it is critical that user sessions and API keys do not share the same credentials

- 
- API keys are never invalidated after a specified amount of time
 - API keys should be configurable to be invalid after a specified amount of time or at a given date and time
 - The user is allowed to specify the API key themselves
 - Usually the backend generates an API key for the user for maximum security guarantees of the given API key
 - No fixed prefix enforced on the API key:
 - Using a fixed prefix that is shared by all API keys allows for detecting leaked API keys on the internet
-

Recommendation

Re-implement the API key mechanisms entirely and follow state-of-the-art recommendations for safe API key handling. Potentially, trusted and verified libraries could be used to ensure that the API key implementation is actually secure. This is especially critical as these implementations typically make use of cryptographic concepts and algorithms that are harder to implement securely.

Status

This finding has been addressed in commit IDs [8360e9b](#), [9ff80a5](#), [2e8b7cf](#), [d439c66](#), [12dfbc2](#)

[TS25] Not using synced blockchain state in database

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The backend synchronizes the blockchain state in a database. This allows for fast querying of blockchain data without involving the Stellar API.

In `src/stellar-contract/escrow/shared/shared-escrow.service.ts:246`, the function `simulateReadOnlyCall` is used that uses the Stellar API to simulate a transaction, see `src/stellar-contract/shared/stellar-transaction-builder.service.ts:255`. This data is used in `src/stellar-contract/escrow/shared/shared-escrow.service.ts:56` to query the attribute `decimals` of a smart escrow from blockchain state.

This can be optimized by using the synchronized state in the database instead. Additionally, such an approach does not prevent race conditions or front-running attacks. Instead, the backend must rely on the smart escrow smart contract to prevent any front-running attacks and submit data in a transactional ordering. Therefore, the database state can be used here.

Recommendation

Replace all non-critical occurrences of querying blockchain state with database queries to synchronized blockchain state in the database. It must be ensured that the database state is synchronized properly with the blockchain.

Status

This finding has been addressed in commit ID [4298787](#).



[TS26] API provided values are silently overridden with default values

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

The backend defines DTO (Data Transfer Objects) that declare the fields and types that can be sent to individual endpoints, typically encoded as JSON in the HTTP body. These values provided by the client are usually transformed and validated in the backend.

Some DTOs fields are silently overridden with default values, e.g.:

- `src/utils/parse.utils.ts:65`
- `src/utils/parse.utils.ts:115`

Recommendation

Change the endpoints and DTOs to either drop fields that are not required by the backend, or alternatively make use of the currently unused fields and values in the DTOs.

Status

This finding has been addressed in commit ID [0843d1b](#).

[TS27] Login authentication procedure does not authenticate user

Severity: High

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

The backend does not have an endpoint dedicated for user logins that authenticates the respective user and returns a session JWT. Instead, the backend provides the endpoints

`/auth/request-api-key` , `/user/:id` , and `/user/create` .

A user can be created with the endpoint `/user/create` without any authentication. Subsequently, a JWT token can be generated and retrieved with the endpoint `/auth/request-api-key` without any authentication (and also no JWT is required to access this endpoint). Subsequently, this JWT can be retrieved with the endpoint `/user/:id` without authentication. These critical endpoints are missing authorization, as described in [\[TS03\] Missing Authentication Guards on Critical Endpoints](#).

It should be noted that the backend generated API keys are valid JWT tokens that can be used to successfully authenticate with the backend, as described in [\[TS24\] API key does not meet security standards](#).

Generally, anybody can log in, generate, and retrieve API keys that are valid JWT tokens for authorization, without providing any authentication. Only the respective public wallet address must be provided to the API.

```
// src/auth/auth.controller.ts:11
@Post("request-api-key")
async login(@Body() loginUserDto: LoginUserDto) {
  return await this.authService.login(loginUserDto);
}

// src/auth/auth.service.ts:59
async login(loginUserDto: LoginUserDto) {
  const { wallet } = loginUserDto;

  if (!wallet) {
    throw new UnauthorizedException("Wallet is required");
}
```

```
const firestore = this.firebaseioService.getFirestore();
const usersCollection = firestore.collection("users");
const user = await usersCollection.doc(wallet).get();

if (user.data() === undefined)
    throw new UnauthorizedException("User are not registered");

const token = this.getJwtToken({ wallet: user.data().address });

await usersCollection.doc(wallet).update({
    apiKey: FieldValue.arrayUnion(token),
});

// eslint-disable-next-line @typescript-eslint/no-unused-vars
const { id, ...userData } = user.data();
return userData;
}
```

Recommendation

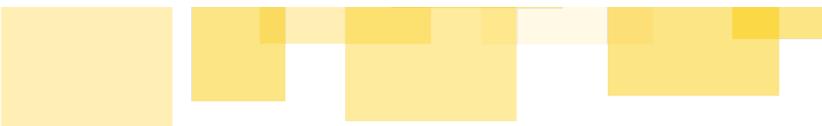
A proper authentication procedure must be implemented and mandatory for all endpoints that require authentication such as login and register endpoints.

As users login with their respective wallet, they must authenticate that they are the respective owner of the wallet. This can be properly implemented by following the SEP-10 standard (<https://github.com/stellar/stellar-protocol/blob/master/ecosystem/sep-0010.md>), as described in the Stellar documentation (<https://developers.stellar.org/docs/build/apps/wallet/sep10>).

Additionally, the generated JWT tokens should not be saved in the database. Instead, whenever the user loses the JWT or the JWT becomes invalid, the user is expected to re-authenticate to get a new JWT.

Status

This finding has been addressed in commit ID [8360e9b](#), [2e8b7cf](#), [4f7b37e](#).



[TS28] The backend relies on data in pendingWriteQueue and getTransaction instead of events and blockchain storage

Severity: High

Difficulty: Medium

Recommended Action: Fix Design

Not addressed by client

Description

The backend provides an API that can be used to fetch smart escrow data from a database instead of the blockchain. The backend ensures that the database is always in sync with the blockchain state. This is currently implemented by requiring all smart escrow creation and mutation transactions to be created by the backend.

The backend stores the data related to the respective transaction in a queue and sends the transaction to the user's client. After the user signed the transaction, the user can either send the signed transaction to the backend, where it is submitted to the blockchain and included in the database, or submit the transaction themselves and only send the respective transaction hash to the backend, where the respective related data is taken from the queue and synchronized with the database. In case the backend receives the transaction hash from the client, it validates whether the transaction was actually submitted to and processed by the blockchain, though only for smart escrow deployment transactions as already described in [TS29] Missing validation whether user-submitted transactions were actually submitted to and processed by the blockchain.

When a user does not deploy the smart escrow with the backend, then the smart escrow is not properly synchronized to the database and cannot be later added to the backend database, thereby denying the user the use of the backend API, even though the smart escrow would pay fees to Trustless Work for the backend API infrastructure.

Additionally, any smart escrow transaction that was not generated by the backend will not be captured, leading to a state where the backend database is not properly in sync with the blockchain.

The backend utilizes the `getTransaction` API endpoint of the Stellar blockchain to confirm the validity of transactions and also to get the contract id from deployment transactions. The Stellar `getTransaction` endpoint` only stores transactions for a restricted period of time, the default being 24 hours, see <https://developers.stellar.org/docs/data/apis/rpc/api->



[reference/methods/getTransaction](#). This puts the backend at risk of missing transactions in case of downtime or similar.

Recommendation

We recommend to replace the `pendingwriteQueue` design choice with a procedure that queries storage state from smart escrows via the Stellar API and syncs the data with the database. This procedure could be run for smart escrows determined via emitted Soroban events that the backend could periodically query for from the Stellar API. Additionally, an API endpoint could be provided to update the state on a specific given smart escrow, in case the events were missed by the backend due to, e.g., downtime. This is because events are similar to transactions only stored for a restricted amount of time in Stellar API nodes.

This would deliberately also allow the backend to process smart escrows that were not deployed by the backend. Therefore the backend must check that the wasm hash of the smart escrow is valid. Additionally, we recommend to validate the smart escrow storage state for valid storage values. This is because we assume that smart escrows with invalid storage values (bypassing initialization validation) could potentially be deployed by exploiting Soroban's smart escrow upgrade functionality.

Status

This finding will be fixed in the future version of the protocol which will implement a dedicated blockchain indexer.

[TS29] Missing validation whether user-submitted transactions were actually submitted to and processed by the blockchain

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

The function `updateFromTxHash` at `src/indexer/indexer.service.ts:23` is used to process pending transactions stored in the `pendingWriteQueue` that previously were not yet submitted to and processed by the blockchain. This function is therefore called after either the backend submits a transaction or after a user submits a transaction and notifies the backend through its `/indexer/update-from-txhash` endpoint, providing the transaction hash.

In case the transaction deployed a smart escrow, i.e. `pending.type === "SAVE_ESCROW"`, the function `updateFromTxHash` uses `getTransaction` to retrieve the smart contract address. This is expected to fail if the transaction has not yet been processed by the blockchain, though there is no dedicated check that validates this directly.

In the other case, where the transaction is different from deployment, the transaction, as long as it is available in the `pendingWriteQueue`, is processed and the mirrored blockchain state in the database updated. There is no validation that checks whether the transaction has been processed by the blockchain. This means that users can create transactions (except for deployment) with the backend and then use the `/indexer/update-from-txhash` endpoint to cause the backend to process these transactions without the user ever having to submit the transactions to the blockchain.

Additionally, due to the `finally` in `updateFromTxHash` that removes the respective transaction from the queue, the transaction is always removed from the queue, regardless of whether the transaction was processed by the backend. E.g., it is expected that this might happen when a deployment transaction is processed by `updateFromTxHash` without the blockchain yet having processed the transaction.

Recommendation

Add proper validation that the transaction that is being processed in `updateFromTxHash` was actually properly processed by the blockchain with `getTransaction`. Additionally, in case the blockchain did not properly or successfully process the transaction, keep the element in the queue. Of course, as previously discussed in [\[TS01\] In-Memory Queue Storage Causing Data Loss and Scaling Issues](#), there needs to be a mechanism in place that removes elements from the queue after a specified period of time has passed.

Status

This finding has been addressed in commit ID [36ece55](#).

[TS30] Recommendation: use self-hosted Stellar nodes as the Stellar API endpoint

Severity: Informative

Not addressed by client

Description

The backend requires a Stellar API endpoint to properly function. In case this API endpoint is hosted by a third-party, this third-party could feed wrong and potentially malicious data to the backend.

Recommendation

To ensure that data returned by the Stellar API is valid, Stellar nodes could instead be self-hosted. These nodes are then expected to always return valid data, as long as they are hosted securely, properly maintained, and kept up to date.

Status

This recommendation was already being considered by the client, and will be fixed in the future implementation of the project outside of the scope of this audit.

[TS31] `set-trustline` endpoint uses a private key as an argument

Severity: High

Difficulty: High

Recommended Action: Fix Code

Addressed by client

Description

The endpoint `/helper/set-trustline` can be used to set a trustline on a Stellar wallet. To do so, this endpoint takes as argument the private and public keypair of the user. A transaction is prepared that enables the trustline for the user that is subsequently signed on the backend with the provided private key and submitted to the blockchain.

This is inherently dangerous for end-users, as they are not supposed to share their private key with other third-parties. This pattern also diverges from the rest of the backend codebase, where an unsigned transaction is prepared on the backend and then sent to the users client. The user can then decide on their client whether to sign the transaction after reviewing it.

Recommendation

We recommend to refactor this endpoint to send an unsigned transaction to the users client, similar to how the other endpoints in the backend operate. There should be no endpoint in the backend that accepts a users private key as an argument.

Status

This finding has been addressed in commit ID [939242c](#).

[TS32] Issued JWT tokens never expire

Severity: Low

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

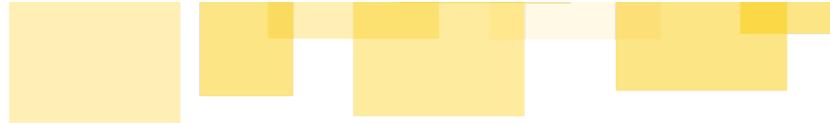
The JWT tokens that are created by the backend never expire. As the users login with their wallet, it is a possibility that the ownership of wallet might change. Therefore, it would be best-practice to require users to periodically re-authenticate.

```
// src/auth/auth.module.ts:13
JwtModule.registerAsync({
  imports: [],
  inject: [],
  useFactory: () => {
    return {
      secret: process.env.JWT_SECRET,
      signOptions: {},
    };
  },
}),
```

Recommendation

Add an expiration date to JWT tokens by including the following option with a proper value:

```
// src/auth/auth.module.ts:13
JwtModule.registerAsync({
  imports: [],
  inject: [],
  useFactory: () => {
    return {
      secret: process.env.JWT_SECRET,
      signOptions: {
        expiresIn: 'x', // update 'x' to a proper value
      },
    };
  },
}),
```



Status

This finding has been addressed in commit ID [8360e9b](#)

[TS33] Potentially never-ending loop querying the Stellar API

Severity: High

Difficulty: High

Recommended Action: Fix Code

Addressed by client

Description

The function `signAndSendTransaction` at `src/utils/transaction.utils.ts:100` can be used to sign a transaction in the backend and subsequently submit that transaction to the blockchain. This function waits until the transaction has been successfully processed by the blockchain.

However, this loop may never terminate in case the blockchain does not process the transaction as expected. In that case, the backend would continuously send a request to the Stellar API every 1000 milliseconds.

```
do {
  await new Promise((resolve) => setTimeout(resolve, 1000));
  getResponse = await server.getTransaction(response.hash);
} while (getResponse.status === "NOT_FOUND");
```

Recommendation

Use a timeout or a maximum number of retries before this loop is terminated and an error is returned.

Status

This finding has been addressed in commit ID [1b3bb0c](#).



Limitations and Recommendations for Follow-Up Audit

Runtime Verification reviewed and tested the codebase over a five-week period, during which a substantial number of issues were identified and reported to the client. Many of these findings have since been addressed; however, several of the implemented fixes involved significant code changes that could not be exhaustively re-audited within the scope and timeframe of this engagement.

While Runtime Verification performed targeted spot-checks of select remediations, we did not conduct a full re-audit of all modified components. Some fixes introduced non-trivial structural changes whose implications and interactions extend beyond the originally defined audit scope. As a result, we cannot guarantee that all remediations have been correctly and securely implemented, nor that new vulnerabilities have not been introduced as a side effect of these changes.

Given the overall volume of code changes and the complexity of the system, we strongly recommend conducting a comprehensive follow-up audit of the updated codebase before the project secures significant value in its smart contracts. A focused re-audit should verify both the correctness of all remediations and the absence of regressions or emergent issues resulting from refactoring.

This recommendation is made in the interest of transparency and due diligence. The audit team's findings and review were limited to the code and commits available during the original audit window, and Runtime Verification cannot assume responsibility for subsequent modifications or unreviewed changes introduced after that period.