Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



tokenforge

Tokenforge Waterfall Distribution



Veridise Inc.

August 24, 2026

Last updated date: Oct 2, 2026

► **Prepared For:**

tokenforge GmbH
www.tokenforge.io

► **Prepared By:**

Evgeniy Shishkin
Mark Anthony

► **Contact Us:**

contact@veridise.com

► **Version History:**

Oct. 2, 2026	V5
Oct. 1, 2026	V4
Oct. 1, 2026	V3
Sep. 28, 2026	V2
Sep. 21, 2026	V1
Sep. 18, 2026	Initial Draft

© 2026 Veridise Inc. All Rights Reserved.

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 5

3 Security Assessment Goals and Scope 6
- 3.1 Security Assessment Methodology . . . 6
- 3.2 Identified Security Risks . . . 6
- 3.3 Scope . . . 7
- 3.4 Classification of Vulnerabilities . . . 8

4 Security Review Assumptions 9
- 4.1 Operational Assumptions . . . 9
- 4.2 Privileged Roles . . . 9

5 Vulnerability Report 12
- 5.1 Detailed Description of Issues . . . 13
  - 5.1.1 V-TWFD-VUL-001: Interest accrual can skip one day . . . 13
  - 5.1.2 V-TWFD-VUL-002: Paid default interest remains outstanding . . . 14
  - 5.1.3 V-TWFD-VUL-003: Tranche token transfers between batches can inflate investor payouts . . . 16
  - 5.1.4 V-TWFD-VUL-004: Migration constraints prevent existing waterfall state from being accurately imported . . . 18
  - 5.1.5 V-TWFD-VUL-005: Current class C interest blocks variable return . . . 21
  - 5.1.6 V-TWFD-VUL-006: Service operator can remove EMT from the waterfall 23
  - 5.1.7 V-TWFD-VUL-007: Repeated accruals within month count common boundary day twice . . . 25
  - 5.1.8 V-TWFD-VUL-008: Instance TTL may expire between monthly calls . . . 27
  - 5.1.9 V-TWFD-VUL-009: Sweep can send tokens to their own contract . . . 28
  - 5.1.10 V-TWFD-VUL-010: Unprotected initialization allows contract takeover . 29
  - 5.1.11 V-TWFD-VUL-011: Service Operator with coinvestor role can stop manifest after approval deadline . . . 32
  - 5.1.12 V-TWFD-VUL-012: Service Operator can close waterfall before all investor payments executed . . . 34
  - 5.1.13 V-TWFD-VUL-013: Pending approvals can apply to different manifest . . 36
  - 5.1.14 V-TWFD-VUL-014: One-step admin rotation can permanently lose control 38
  - 5.1.15 V-TWFD-VUL-015: Duplicate tranche holders can distort pro rata distributions . . . 40
  - 5.1.16 V-TWFD-VUL-016: Code quality improvements . . . 42
  - 5.1.17 V-TWFD-VUL-017: Token addresses can point to payout forwarder . . . 44
  - 5.1.18 V-TWFD-VUL-018: XLM relay can use arbitrary token . . . 46
  - 5.1.19 V-TWFD-VUL-019: Interest rounding does not account for configured EMT decimals . . . 48

 © 2026 Veridise Inc.

A Appendix **50**
A.1 Intended Behavior: Non-Issues of Note 50
A.1.1 V-TWFD-APP-VUL-001: Default interest charges a full month instead of the overdue period 50

 © 2026 Veridise Inc.

About Veridise

Veridise is an independent security firm founded by academics and security researchers with deep expertise in formal methods, programming languages, and applied cryptography. The firm focuses on high-assurance security for blockchain and cryptographic systems, combining rigorous theory with practical adversarial analysis.

Veridise provides full-stack blockchain security services spanning smart contracts, zero-knowledge circuits and proving systems, consensus and execution clients, cryptographic libraries, and supporting infrastructure. The team routinely analyzes systems at the boundary between protocol design, implementation, and cryptography, where failures are both subtle and high-impact. To date, Veridise has conducted hundreds of security assessments for protocols securing billions of dollars in TVL.

Dynamic Security Veridise treats security as an ongoing engineering discipline rather than a one-time validation exercise. Effective security requires iterative review, continuous feedback, and close collaboration between auditors and system designers. Throughout the engagement, Veridise analysts communicated regularly with the tokenforge GmbH team using AuditHub, enabling rapid clarification of design intent, prompt discussion of findings, and efficient validation of fixes.

In parallel with manual review, Veridise maintains active research programs focused on automated and formal analysis of blockchain systems. These efforts have produced tools such as *Vanguard* for static analysis and *Picus* for formal verification, which are available through AuditHub. AuditHub integrates these techniques directly into CI/CD workflows, supporting continuous vulnerability detection and regression analysis beyond the scope of a single audit.

Additional details and case studies are available at `https://audithub.dev/case-studies/`.

Veridise Reports Veridise reports are published in the public audit archive linked below. Only reports obtained from this archive or confirmed directly by a representative of Veridise should be considered official Veridise reports.

`https://veridise.com/audits-archive/`

© 2026 Veridise Inc.

1 Executive Summary

From Aug. 24, 2026 to Sep. 10, 2026, tokenforge GmbH engaged Veridise to conduct a security assessment of the Tokenforge Waterfall Distribution protocol. The security assessment covered the Waterfall token distribution contract, the Permission55 access control contract, and the Payout Forwarder payment contract. Veridise conducted the assessment over 28 person-days, with 2 security analysts reviewing the project over 14 days on commit `809af83`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough manual code review.

Project Summary. Waterfall is an on-chain system for distributing funds to investors in a financing deal. It tracks what is owed to three investor classes—A, B, and C—and pays them in a defined order. Class A is paid first, followed by Class B and then Class C. Payments are made in EMT*, and a reserve can help cover Class A interest when the current collection is insufficient.

The protocol consists of the following components:

- **Waterfall** contract manages distribution cycles. It tracks principal and interest, holds the reserve, calculates how incoming funds should be used, and pays investors based on their tranche-token holdings.
- **Permission55** contract manages access rights. Waterfall checks it to determine who may submit a distribution plan, approve it, execute payment batches, or carry out other restricted actions. Permission55 does not calculate or make investor payments.
- **Payout Forwarder** receives EMT on behalf of recipients who cannot receive payments directly from Waterfall. An authorized operator then forwards the funds to a configured destination. It has its own administrator and operator list; it does not use Permission55 for access control.

Users of the protocol are grouped by their respective incentives:

- **Investors** hold Class A, B, or C tranche tokens and expect to receive the payments owed to their class.
- **Service Operator** runs the day-to-day process, submits distribution plans, and manages the contract's operating state and reserve.
- **Batch Operator** executes the approved payment batches. It is the only role authorized to trigger batch execution, with each batch performing at most `batch_size` EMT transfers.
- **Class A and Class B representatives** review distribution plans and proposed changes to protect their respective investors' interests.
- **Administrators** set up the contracts and manage the roles recorded in Permission55 so that restricted actions remain available only to authorized participants.

A typical distribution cycle proceeds as follows:

1. **Off-chain preparation.** The parties calculate the funds available for distribution, account for relevant expenses, and agree on a distribution plan. The resulting information is recorded in a manifest.

* EMT (e-money token) refers to the euro-backed token used for payments in Waterfall. The intended asset is AllUnity's EURAU, issued on Stellar as a SEP-41 token and redeemable 1:1 for euros.

© 2026 Veridise Inc.

2. **Contract funding.** The required EMT is transferred to the Waterfall contract. The Service Operator may also fund the reserve used to cover eligible Class A interest shortfalls.
3. **Manifest submission.** The Service Operator submits the manifest details and its hash to the Waterfall contract, creating a new distribution cycle.
4. **Manifest approval.** The Class A and Class B Co-Investor representatives review the submitted manifest. The normal execution path requires approval from at least two distinct representatives of each class. An eligible representative may stop the manifest before their side reaches its approval threshold. If the approval deadline expires without the required approvals, the protocol can instead use its timeout execution path.
5. **Waterfall calculation.** Once a manifest is approved, the Batch Operator may call `execute_batch()`. The first call accrues interest and applies the available EMT according to the payment priority: expenses and reserve requirements are handled first, followed by payments owed to Class A, Class B, and Class C in their required order. The contract records an execution plan for the resulting investor payments.
6. **Batched investor payments.** Further calls to `execute_batch()` process the execution plan in batches. Only the Batch Operator may trigger batch execution, and each batch performs at most `batch_size` EMT transfers. EMT assigned to each tranche is distributed among its investors in proportion to their tranche-token holdings.
7. **Cycle completion.** After the final batch, the contract marks the distribution as complete and preserves the updated principal, interest, default-interest, and reserve balances for the next cycle. Investors receive their EMT directly or through their configured payout addresses.

Code Assessment. The developers provided the source code for the Waterfall, Permission55, and Payout Forwarder contracts, along with documentation on the system's data flows, roles, payment rules, and operating procedures. They also walked the Veridise security analysts through the protocol, focusing on its more complex flows. The test suite covered initialization, access control, approvals, interest calculations, reserve management, batched distributions, and the contract lifecycle. It also included tests for Permission55 and Payout Forwarder, as well as a multi-cycle test comparing Waterfall calculations with reference investor-report figures.

Summary of Issues Detected. The security assessment uncovered 19 issues, 4 of which are assessed to be of high severity by the Veridise analysts. The high severity findings include V-TWFD-VUL-001 where interest accrual across month boundaries can skip up to one day of interest, V-TWFD-VUL-002 which notes that paid default interest remains included in the aggregate outstanding balance, causing already paid debt to remain due, V-TWFD-VUL-003 that shows how tranche token transfers between payout batches can cause the same tokens to contribute towards multiple payouts, allowing distributions to exceed their allocated amounts and V-TWFD-VUL-004 which describes how migration constraints can prevent existing waterfall state from being accurately imported on-chain.

The Veridise analysts identified 3 medium-severity issues, including V-TWFD-VUL-005 where current period interest can incorrectly block Class C variable returns, V-TWFD-VUL-006 which shows how the Service Operator can sweep legitimate EMT that should remain available for waterfall distributions, and V-TWFD-VUL-007 which identifies how repeated accruals within a month can double count shared boundary days and overstate interest.

The Veridise analysts also identified 8 low-severity issues, some of which are: V-TWFD-VUL-010 shows how unprotected initialization can allow deployed contracts to be taken over before legitimate initialization, V-TWFD-VUL-012 highlights how the waterfall can be closed while

investor payouts are still pending leaving investor funds stranded, V-TWFD-VUL-013 describes how delayed approvals can apply to a different manifest than the one originally reviewed, V-TWFD-VUL-015 identifies duplicate tranche holders that can distort pro rata distributions, and V-TWFD-VUL-014 notes how one-step administrator rotation can permanently remove administrative control. The security assessment also identified 4 warnings.

tokenforge GmbH has addressed all of the raised issues. In particular, following discussions with the client and further review, V-TWFD-APP-VUL-001 has been confirmed as intended behavior, while all remaining issues have been fixed.

After the project had concluded, the team returned with one further change (commit 2793313), implementing the requirement that no transaction[†] may contain more than `batch_size` EMT token transfers. Each cycle's payouts are now frozen into a fixed execution plan at batch 0, and every `execute_batch()` call pays at most `batch_size` consecutive entries, one transfer each. A new Batch Operator role was also introduced as the only role authorized to execute these batch operations, preventing other parties from stacking batch calls. The Veridise analysts reviewed the changes and did not identify any issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Tokenforge Waterfall Distribution.

Establish authoritative sources for protocol behavior. The project's extensive documentation is a strength: it shows that many aspects of the protocol were considered before implementation. However, some documents give conflicting or imprecise accounts of expected behavior, including role responsibilities and payout calculations. This makes it difficult to determine whether a difference between the code and a document is a defect or simply reflects an outdated description.

We recommend establishing a small, clearly identified set of authoritative references for the protocol's behavior. These should define which source takes precedence when documents disagree, especially where the legal terms and the calculation spreadsheet govern different aspects of the waterfall. Other documents should be aligned with those references or clearly marked as explanatory. This would give developers and reviewers a consistent basis for checking the implementation and tests, and make future changes easier to assess.

Test the migration from an existing deal. The developers explained that the Waterfall will be deployed as a continuation of an existing business process. It will therefore begin with financial history, rather than with the full principal and no accrued interest. The existing six-cycle test is a useful check against investor reports, but migration testing should also cover different states the deal could have reached by the deployment date. We recommend testing at least the following scenarios:

- A tranche has already repaid part of its original principal. Check that the imported state preserves both the original principal and the smaller outstanding balance.
- A tranche has unpaid interest from completed months and interest accrued during the current month. Check that the two amounts remain distinct and that the next accrual neither skips nor counts any days twice.
- A tranche has unpaid default interest and multiple interest amounts with different due dates. Check that the amounts, due dates, and accrual history survive migration, including when a payment was made shortly before the cutover.

[†] `close()` is a documented exception from this rule.

- One or more tranches have been partly or fully repaid, while others still have outstanding principal or interest. Check that the next distribution follows the correct payment priority.
- The contract receives the first collection after deployment. Check the complete path through accrual, approval, batched investor payments, reserve use, and final balances against an independent calculation or investor report.
- The imported holder lists, payout addresses, token balances, and reserve funding match the agreed cutover records. Check that the first distribution pays the intended investors and that its funding is available.
- The supplied state is inconsistent—for example, its interest components do not reconcile, a due date is invalid, or its timestamp is later than deployment. Check that initialization rejects it without leaving a partially initialized contract.

These should be tests of the public initialization and distribution flow, using representative cutover records, not tests that insert state directly into storage. Some of these scenarios are not supported by the current initialization rules: for example, `initialize()` requires outstanding principal to equal initial principal and always starts the recorded reserve balance at zero.

The tests should make such limitations visible so they can be resolved before production migration, rather than adjusting the cutover figures merely to make initialization succeed.

Migration during an unfinished payout. Test a cutover after batch 0 and after each subsequent batch, while some investors have been paid and others have not. Verify that the new contract can preserve the remaining investor payments, the execution plan and next-batch position, tranche and reserve balances, and the EMT needed to finish the payout. The test should confirm that no investor is paid twice or omitted, including if the cutover fails partway through. If migration at this point is not supported, document that limitation explicitly and require the current payout cycle to finish before migration begins.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Tokenforge Waterfall Distribution	`809af83`	Rust	Stellar

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 24–Sep. 10, 2026	Manual	2	28 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	4	4	4
Medium-Severity Issues	3	3	3
Low-Severity Issues	8	8	8
Warning-Severity Issues	4	4	4
Informational-Severity Issues	0	0	0
TOTAL	19	19	19

Table 2.4: Category Breakdown.

Name	Number
Logic Error	8
Data Validation	7
Access Control	2
Authorization	1
Maintainability	1

© 2026 Veridise Inc.

3 Security Assessment Goals and Scope

3.1 Security Assessment Methodology

The security assessment process consists of the following steps:

1. Gather an initial understanding of the protocol's business logic, users, and workflows by reviewing the provided documentation and consulting with the developers.
2. Identify all valuable assets in the protocol.
3. Identify the main workflows for managing these assets.
4. Identify the most significant security risks associated with these assets.
5. Systematically review the codebase for execution paths that could trigger the identified security risks, considering different assumptions.
6. Prioritize one finding over another by assigning a severity level to each.

To improve the depth and efficiency of the review, analysts used the following tools:

- *LLM-assisted review.* Security analysts performed a comprehensive manual review of all in-scope components, including detailed examination of the source code and system design. Large language models (LLMs) were used as an auxiliary aid to support analyst workflows, including structured reasoning, protocol summarization, trust-boundary mapping, hypothesis generation, proof-of-concept development, and report drafting. All findings, conclusions, and supporting evidence were independently reviewed and validated by analysts against the source code before inclusion in the final report.

3.2 Identified Security Risks

After the initial phase of the security assessment was completed, a list of potential security risks was generated. Security analysts used this list during the code review as a starting point to identify potential attack vectors. A few of these risks, when expressed as questions, include the following:

- Are approvals bound to one specific manifest and distribution cycle, or can they be reused after the manifest changes?
- Can the timeout path distribute an amount that differs from what the approvers expected or from the EMT actually available?
- Can EMT sent directly to Waterfall be mistaken for a reserve deposit, a current collection, or excess funds?
- Can a reserve withdrawal or sweep remove EMT needed for investor payments that have been calculated but not yet sent?
- Can interrupted, repeated, or out-of-order batch execution cause an investor to be paid twice, skipped, or paid the wrong amount?
- Can duplicate or missing addresses in a tranche holder list distort the distribution of payments?
- Can changes in tranche-token balances between payment batches affect investors' shares of the same distribution?

© 2026 Veridise Inc.

- Can a configuration change during an active cycle alter payment destinations, interest calculations, or batch processing after the cycle has been approved?
- Can incorrect token addresses, token decimals, or unexpected behavior by an external token contract cause Waterfall to miscalculate or fail to make payments?
- Can interest accrual, rounding, or partially repaid principal cause the contract to overpay one class or pay a junior class before amounts due to a senior class are settled?
- Can inconsistent values supplied during initialization or migration create obligations that the contract cannot account for correctly?
- Can pausing or closing Waterfall during an active distribution leave payments unfinished or release funds that are still owed?
- Can a Payout Forwarder operator, administrator, or configuration change redirect EMT away from its intended recipient?
- Does the time-to-live (TTL) extension logic correctly reflect the expected workflow?
- Is there an initialization front-running potential in the contracts?

3.3 Scope

The scope of this security assessment is limited to a specific set of source files from the repository at commit `809af83`, as agreed upon with the Tokenforge Waterfall Distribution developers:

- `contracts/payout-forwarder/src/errors.rs`
- `contracts/payout-forwarder/src/events.rs`
- `contracts/payout-forwarder/src/lib.rs`
- `contracts/payout-forwarder/src/storage.rs`
- `contracts/payout-forwarder/src/types.rs`
- `contracts/permission55/src/admin.rs`
- `contracts/permission55/src/balance.rs`
- `contracts/permission55/src/errors.rs`
- `contracts/permission55/src/events.rs`
- `contracts/permission55/src/holders.rs`
- `contracts/permission55/src/lib.rs`
- `contracts/permission55/src/permission_roles.rs`
- `contracts/permission55/src/permission_sets.rs`
- `contracts/permission55/src/registry.rs`
- `contracts/permission55/src/storage_types.rs`
- `contracts/waterfall/src/calendar.rs`
- `contracts/waterfall/src/distribution.rs`
- `contracts/waterfall/src/errors.rs`
- `contracts/waterfall/src/events.rs`
- `contracts/waterfall/src/interest.rs`
- `contracts/waterfall/src/lib.rs`
- `contracts/waterfall/src/manifest.rs`
- `contracts/waterfall/src/roles.rs`
- `contracts/waterfall/src/state_machine.rs`
- `contracts/waterfall/src/storage.rs`
- `contracts/waterfall/src/types.rs`
- `contracts/waterfall/src/waterfall.rs`

3.4 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Security Review Assumptions

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Tokenforge Waterfall Distribution.

- The contract addresses, payment destinations, and financial parameters supplied at deployment are checked against approved deal records before initialization.
- Any tranche state imported from an existing deal accurately reflects the principal, interest, and reserve balances at the time of migration.
- The holder lists and payout addresses supplied to Waterfall identify the intended recipients, while tranche-token balances reflect investors' actual holdings.
- Permission55 roles are issued to the intended people. Its administrator and role-issuance keys, as well as the Payout Forwarder's administrator key, are kept under appropriate custody controls.
- Class A and Class B approvers act as separate decision-making groups. Before approving a manifest, they compare its on-chain hash and amounts with the document and financial records they reviewed.
- The off-chain collection, expense, and conversion processes provide accurate figures for each manifest. The EMT needed for a distribution is delivered to Waterfall before execution.
- The supplied legal and accounting materials describe the business rules intended by the deal parties. The code review does not independently establish the parties' legal interpretation of those materials.
- Operators normally run distributions on the expected monthly schedule and monitor unfinished batches and contract storage so that prolonged inactivity does not prevent further operation.
- If an investor payment fails because a token issuer has restricted a recipient, the responsible parties resolve the restriction or approve a corrected payout address off-chain before retrying the batch.
- Payout Forwarder destinations are checked before use. Forwarder operators release funds to those destinations as intended, and its administrator uses the emergency sweep only under the agreed operational controls.

4.2 Privileged Roles

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensitivity.

Highly-privileged roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised. Time-sensitive *emergency*

© 2026 Veridise Inc.

roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assumed that the operators of privileged roles perform their assigned responsibilities as intended. A role holder's deliberate misuse of powers that the protocol intentionally grants to that role was not treated as a contract-level vulnerability. Whether the contracts correctly enforce the intended limits of each role remained in scope.

- Highly-privileged, emergency roles:
  - **Waterfall Service Operator** can pause or resume the contract, stop an eligible manifest, manage reserve funds, sweep tokens, and close the contract. The same role also submits manifests and proposes configuration changes during normal operations.
  - **Payout Forwarder administrator** can use the emergency sweep to transfer any token held by the forwarder, including EMT, to a chosen address. This power remains available even when the normal forwarding destination is locked.
  - **Permission55 burner** can revoke role tokens across permission sets, including roles used by Waterfall. This may be necessary to respond quickly to a compromised role holder.
- Highly-privileged, non-emergency roles:
  - **Waterfall initialization signer** supplies the initial token addresses, financial state, holder lists, and payment destinations. Initialization is a one-time action; the signer is not checked against a pre-existing Permission55 role.
  - **Permission55 administrator** can appoint minters and burners, manage permission sets, and transfer the administrator position.
  - **Permission55 minter** can issue any role token in any permission set. A compromised minter could assign Waterfall roles to attacker-controlled addresses.
- Limited-authority, emergency roles:
  - **Individual Class A and Class B representatives** can stop a manifest before execution while their own group's approval quorum has not been reached. A representative can also cancel a pending configuration change.
  - **Waterfall Batch Operator** is the only role authorized to execute investor payment batches through `execute_batch()`. Each call processes the fixed execution plan and may perform at most `batch_size` EMT transfers.
- Limited-authority, non-emergency roles:
  - **Payout Forwarder whitelisted operators** can release EMT held by the forwarder to its configured destination. They cannot choose a different EMT recipient through the normal forwarding function.
  - **Permission55 per-set administrators** can manage the whitelist, blacklist, or contract registry of their assigned permission set. These roles do not grant the ability to mint or burn Waterfall roles.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the codebase, and its current status (i.e., acknowledged, fixed, etc.) are specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-TWFD-VUL-001	Interest accrual can skip one day	High	Fixed
V-TWFD-VUL-002	Paid default interest remains outstanding	High	Fixed
V-TWFD-VUL-003	Tranche token transfers between batches . . .	High	Fixed
V-TWFD-VUL-004	Migration constraints prevent existing . . .	High	Fixed
V-TWFD-VUL-005	Current class C interest blocks variable return	Medium	Fixed
V-TWFD-VUL-006	Service operator can remove EMT from . . .	Medium	Fixed
V-TWFD-VUL-007	Repeated accruals within month count . . .	Medium	Fixed
V-TWFD-VUL-008	Instance TTL may expire between monthly . . .	Low	Fixed
V-TWFD-VUL-009	Sweep can send tokens to their own contract	Low	Fixed
V-TWFD-VUL-010	Unprotected initialization allows contract . . .	Low	Fixed
V-TWFD-VUL-011	Service Operator with coinvestor role can . . .	Low	Fixed
V-TWFD-VUL-012	Service Operator can close waterfall before . . .	Low	Fixed
V-TWFD-VUL-013	Pending approvals can apply to different . . .	Low	Fixed
V-TWFD-VUL-014	One-step admin rotation can permanently . . .	Low	Fixed
V-TWFD-VUL-015	Duplicate tranche holders can distort pro . . .	Low	Fixed
V-TWFD-VUL-016	Code quality improvements	Warning	Fixed
V-TWFD-VUL-017	Token addresses can point to payout forwarder	Warning	Fixed
V-TWFD-VUL-018	XLM relay can use arbitrary token	Warning	Fixed
V-TWFD-VUL-019	Interest rounding does not account for . . .	Warning	Fixed

© 2026 Veridise Inc.

5.1 Detailed Description of Issues

5.1.1 V-TWFD-VUL-001: Interest accrual can skip one day

Severity	High	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/waterfall/src/interest.rs:185-194`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/9		

Description The interest calculation can skip part of a day when an accrual period crosses a month boundary.

The problem is in this code:

```
let day_after = cursor + calendar::SECONDS_PER_DAY;
calendar::days_between(day_after, next_month)
```

The code assumes that adding `SECONDS_PER_DAY` moves `cursor` to the start of the next calendar day. This only works when `cursor` is exactly at midnight. If `cursor` is at noon, adding 24 hours moves it to noon on the next day, not to the start of that day.

For example, if `cursor` is January 15 at 12:00, `day_after` becomes January 16 at 12:00. The remaining time before February 1 is 15.5 days. If `days_between` rounds this down to 15, part of January 16 is skipped. January is then treated as having 30 accrued days instead of 31.

Tests may not catch this because they use timestamps set to midnight.

Impact The contract can calculate less interest than expected. The difference is up to one day of interest for each affected month boundary.

This can produce incorrect tranche balances and payment amounts. The financial difference grows with the principal and interest rate.

Recommendation Move `day_after` to the start of the next calendar day instead of adding 24 hours to the original timestamp:

```
let day_start = cursor - (cursor % calendar::SECONDS_PER_DAY);
let day_after = day_start + calendar::SECONDS_PER_DAY;
let days = calendar::days_between(day_after, next_month);
```

It is recommended to add a test where `cursor` has a non-midnight time and the calculation crosses a month boundary.

Developer Response The developers have implemented the recommendation.

5.1.2 V-TWFD-VUL-002: Paid default interest remains outstanding

Severity	High	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/waterfall/src/interest.rs:242-243`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/10		

Description Default interest is added to both `default_interest_outstanding` and the aggregate `interest_outstanding` balance:

```
tranche.default_interest_outstanding += period_default;
tranche.interest_outstanding += period_total_note + period_default;
```

This is consistent with treating `default_interest_outstanding` as a separate component of the total interest balance. However, the payment logic does not keep the two fields synchronized.

When default interest is paid, both `apply_interest_payment` and `pay_interest_then_principal` reduce only `default_interest_outstanding`:

```
tranche.default_interest_outstanding -= default_paid;
```

Neither function subtracts `default_paid` from `interest_outstanding`. As a result, paid default interest remains included in the tranche's aggregate interest balance.

The difference grows every time default interest accrues and is paid.

Impact A tranche can appear to have unpaid interest after its default interest has been fully paid.

This can produce incorrect outstanding balances, block later waterfall payments, and prevent the contract from treating the tranche as fully settled. Users may receive incorrect payment amounts or be asked to cover debt that has already been paid.

Recommendation Update both `apply_interest_payment` and `pay_interest_then_principal` so that every default interest payment reduces both `default_interest_outstanding` and the aggregate `interest_outstanding`.

Apply this logic in both functions:

```
let default_paid =
    core::cmp::min(remaining, tranche.default_interest_outstanding);

tranche.interest_outstanding -= default_paid;
tranche.default_interest_outstanding -= default_paid;
```

It is recommended to add tests for both the Class A and Class B payment paths. Each test should accrue default interest, pay it in full, and verify that `default_interest_outstanding` becomes zero and that the paid default interest is no longer included in `interest_outstanding`.

This keeps the aggregate interest balance synchronized with its default interest component.

Developer Response The developers have implemented the recommendation.

5.1.3 V-TWFD-VUL-003: Tranche token transfers between batches can inflate investor payouts

Severity	High	**Commit**	809af83
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/waterfall/src/distribution.rs:72-181`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/26		

Description When a waterfall distribution begins, the execution plan snapshots the number of token holders for each tranche, as well as the total amount allocated to that tranche. These values are snapshotted when the first investor batch is executed, and they remain constant throughout the remainder of the execution plan.

```
// Cache holder lists and sum_balances per tranche.
// Always load holders even if tranche total is 0 (we need addresses for the flat index).
let a_holders = registry.class_a_holders.clone();
let b_holders = registry.class_b_holders.clone();
let c_holders = registry.class_c_holders.clone();

// Only compute sum_balances for tranches with payments (avoids unnecessary token calls)
let a_sum = if plan.class_a_total > 0 {
   sum_balances(env, &a_holders, &config.class_a_token)
} else {
   0i128
};
let b_sum = if plan.class_b_total > 0 {
   sum_balances(env, &b_holders, &config.class_b_token)
} else {
   0i128
};
let c_sum = if plan.class_c_total > 0 {
   sum_balances(env, &c_holders, &config.class_c_token)
} else {
   0i128
};
```

Snippet 5.1: Snippet from `execute_investor_batch()` in `distribution.rs`

Investors are then paid out over multiple calls to `execute_batch()`, with the first batch already being processed during the initial call. The amount paid to each token holder is calculated based on their pro rata share of the tranche:

```
(total_amount * holder_balance) / sum_balances
```

However, while `total_amount` and `sum_balances` are determined during the initial batch and remain fixed throughout the execution plan, the `holder_balance` is read when the holder's batch is processed. This is problematic because token balances can change between batches while the values used as the basis for the overall waterfall distribution remain fixed.

As a result, a token holder who has already received their distribution can transfer their tranche tokens to another holder whose batch has not yet been processed. The receiving holder's

increased balance will then be used when calculating their payment, allowing the same tranche tokens to effectively contribute multiple times towards the payout.

For example, assume tranche A contains Alice and Bob, each holding `50` tranche tokens, and `100` EMT is allocated to tranche A. For simplicity, assume the execution batch size is `1`. During the first batch, Alice's `50` tokens entitle her to `50` EMT. The values used by the execution plan remain based on the original `100` token supply and `100 EMT` allocation.

After receiving her payment, Alice can transfer her `50` tranche tokens to Bob, increasing Bob's balance from `50` to `100`. When Bob is processed in the next batch, his current balance is used in the pro rata calculation:

```
(100 * 100) / 100 = 100 EMT
```

The distribution therefore pays `50 EMT` to Alice and another `100 EMT` to Bob, for a total payout of `150 EMT`, even though only `100 EMT` was allocated to the tranche. Without the transfer, Bob would have received only `50 EMT`. Moreover, Bob can later transfer the tokens to another holder, such as Charlie, whose batch has not yet been processed. This allows the same tokens to continue contributing towards additional payouts across subsequent batches.

This is possible because individual holder balances are not fixed when the distribution begins, even though the other values used as the basis for the pro rata distribution are. Additionally, the execution does not ensure that the cumulative amount paid to tranche A holders remains bounded by the snapshotted `class_a_total`.

Impact Tranche tokens can be transferred between distribution batches to cause the same tokens to contribute towards multiple payouts. This can cause a tranche to distribute more assets than allocated and allow investors to receive more than their intended share of the waterfall distribution, breaking the core invariant that an investor should not receive more than they are owed.

Since the overall collection amount remains unchanged, overpaying investors in earlier batches can also leave insufficient funds to pay investors processed later in the execution plan, preventing the distribution from completing as intended.

Recommendation Snapshot each holder's tranche token balance when the waterfall distribution begins and use the snapshotted balances for all subsequent payout calculations. Additionally, ensure that the cumulative amount distributed to each tranche cannot exceed the amount allocated to that tranche in the execution plan.

Alternatively, consider making the tranche investor tokens non-transferable for the duration of the distribution. This would prevent token balances from being moved between holders while the batched payout is in progress.

Developer Response The developers have implemented the recommendation.

5.1.4 V-TWFD-VUL-004: Migration constraints prevent existing waterfall state from being accurately imported

Severity	High	**Commit**	809af83
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/waterfall/src/lib.rs:245-253`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/21		

Description When the waterfall is initialized, the tranche accounting can either start from a fresh state or carry forward the state of an existing deal through migration. A fresh initialization begins with the full principal outstanding and no previously accrued interest. However, when migrating an existing deal, the supplied tranche state needs to capture how the deal has progressed before being moved on-chain, including any principal that has already been repaid and interest that has already accrued.

In this particular scenario, the waterfall is expected to be initialized through the migration path after operating off-chain for some time. The principal and interest values supplied during initialization therefore need to accurately carry the existing accounting state into the contract.

However, the validation performed during `initialize()` imposes several constraints that do not fully account for this progressed state. In particular, it assumes that the entire initial principal remains outstanding and imposes an interest accounting relationship that excludes accrued default interest.

```
pub fn initialize(...)
{
 // veridise: elided

 // At initialization, principal_outstanding must equal initial_principal.
 // Interest: either zero (fresh start) or non-zero when last_accrual_ts is set
 // (migration - carry over interest from off-chain / previous contract).
 for (_, t) in [
     ("class_a", &initial_tranches.class_a),
     ("class_b", &initial_tranches.class_b),
     ("class_c", &initial_tranches.class_c),
 ] {
     if t.principal_outstanding != t.initial_principal {
         return Err(WaterfallError::InvalidAmount);
     }
     let migration = t.last_accrual_ts != 0;
     if !migration && t.interest_outstanding != 0 {
         return Err(WaterfallError::InvalidAmount);
     }
     if migration {
         let expected = t
             .completed_period_interest
             .checked_add(t.period_accrual)
             .ok_or(WaterfallError::InvalidAmount)?;
         if t.interest_outstanding != expected {
             return Err(WaterfallError::InvalidAmount);
         }
     }
```

```
29      if t.interest_outstanding < 0
30          || t.completed_period_interest < 0
31          || t.period_accrual < 0
32      {
33          return Err(WaterfallError::InvalidAmount);
34      }
35  }
36 }
```

Snippet 5.2: Snippet from `initialize()` in `contracts/waterfall/src/lib.rs`

First, `principal_outstanding` is required to be equal to `initial_principal`. This does not account for principal that may already have been repaid before migration. For example, if a tranche initially contained `1,000` principal and `300` had already been repaid, the migrated state should be able to represent an initial_principal of `1,000` and a principal_outstanding of `700`. The current validation rejects such a state. This can be worked around by supplying the reduced `700` value as both `initial_principal` and `principal_outstanding`. However, doing so causes `initial_principal` to represent the principal remaining at migration rather than the actual initial principal of the tranche.

Additionally, when `last_accrual_ts` is nonzero, `initialize()` treats the tranche as a migration and requires `interest_outstanding` to equal the sum of `completed_period_interest` and `period_accrual`. Although `initialize()` allows a non-zero `default_interest_outstanding`, its consistency check does not include that amount in `interest_outstanding`. This is inconsistent with the runtime accrual logic, where accrued default interest is also added to `interest_outstanding`. Consequently, an existing tranche with accrued default interest cannot be migrated while preserving the same relationship between its accounting fields that would exist if the interest had accrued on-chain.

Together, these constraints can require the migrated tranche values to be modified to satisfy initialization rather than accurately representing the waterfall state that existed immediately before migration.

Impact The migration process may be unable to accurately represent the existing state of a waterfall that has already undergone principal repayments or accrued default interest. This can either prevent the existing state from being migrated or require accounting values to be modified to satisfy the initialization constraints, causing the on-chain tranche state to differ from the state being migrated.

Recommendation Use separate validation rules for fresh deployments and migrated deals:

- For a fresh deployment, continue to require the full initial principal to remain outstanding and all interest balances to be zero.
- The migration validation should also follow the same accounting relationship used by the runtime logic. The aggregate interest balance should equal the sum of completed-period interest, current-period accrual, and outstanding default interest.

Add migration tests covering:

- a tranche whose principal was partially repaid before migration;
- a fully repaid tranche with zero `principal_outstanding`;
- a tranche with accrued base and default interest;
- rejection when `principal_outstanding` exceeds `initial_principal`;

- rejection when the interest components do not match `interest_outstanding`.

This allows the contract to preserve the original principal and import the current outstanding balances without changing the pre-migration accounting state.

Developer Response The developers have implemented the recommendation.

5.1.5 V-TWFD-VUL-005: Current class C interest blocks variable return

Severity	Medium	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/waterfall/src/waterfall.rs:427-428`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/11		

Description The Class C variable return can be blocked by interest accrued during the current calendar month, even though that interest is not payable yet.

The problem is in the Step 7 condition:

```
let class_c_variable_return = if tranches.class_a.payable_outstanding() == 0
    && tranches.class_b.payable_outstanding() == 0
    && tranches.class_c.total_outstanding() <= residual
    && remaining > 0
```

Step 6 pays Class C obligations using `payable_outstanding()`. This includes principal, completed-period interest, and default interest, but excludes the current `period_accrual`.

Step 7 then checks `total_outstanding()`, which includes `interest_outstanding` and therefore also includes the current `period_accrual`. The two steps use different definitions of the debt that must be settled before variable return can be distributed.

For example, assume Class C has 50 EMT of principal, 1 EMT of payable interest, and 0.02 EMT of current-period interest. If the contract receives 150 EMT, Step 6 pays the 1 EMT of interest and 49 EMT of principal, leaving only the fixed 1 EMT principal residual.

The remaining 100 EMT should be distributed as Class C variable return. Instead, Step 7 sees a total balance of 1.02 EMT because it includes the non-payable current-period interest. The condition fails and the 100 EMT remains in the contract.

Impact Class C variable return can be delayed or left undistributed whenever Class C has a non-zero interest rate and current-period interest has accrued.

Funds that should be paid to Class C holders remain in the contract even after every currently payable obligation has been settled. The affected amount can include the entire collection remaining after Step 6.

Recommendation Use `payable_outstanding()` in the Step 7 Class C residual check, matching the accounting basis already used by Step 6:

```
let class_c_variable_return = if tranches.class_a.payable_outstanding() == 0
    && tranches.class_b.payable_outstanding() == 0
    && tranches.class_c.payable_outstanding() <= residual
    && remaining > 0
```

This allows variable return after all currently payable Class C obligations have been settled, while keeping the current `period_accrual` recorded for a later payment period.

It is recommended to add a regression test with a non-zero Class C interest rate and a positive current `period_accrual`. The test should verify that Step 6 leaves the fixed principal residual and that Step 7 distributes the remaining collection as Class C variable return.

Developer Response The developers now make use of `payable_outstanding()` instead of `total_outstanding()` in the mentioned location.

5.1.6 V-TWFD-VUL-006: Service operator can remove EMT from the waterfall

Severity	Medium	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/waterfall/src/lib.rs:447-459, 1673-1715`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/27		

Description The contract allows the Service Operator to transfer EMT to the configured `sweep_wallet` through `perform_sweep()`.

The legal waterfall document treats a broad range of incoming funds as Available Amounts and requires them to be distributed according to the agreed priority of payments. The contract receives these funds through direct EMT transfers. A transfer itself does not identify the relevant cycle, manifest, source, or purpose, so the contract cannot distinguish an accidental EMT transfer from legitimate Available Amounts by inspecting its token balance.

Despite this, `execute_sweep()` assumes that only two accounting values need to remain in the contract:

1. The current recorded reserve balance; and
2. The original `collection_amount` while a manifest is active and not stopped.

Every other EMT held by the contract is classified as excess and can be swept.

This classification is not valid. An amount does not become unrelated to the waterfall merely because it is not represented by these two counters. Legitimate EMT can fall outside the calculated earmark at several points in the contract lifecycle:

- **Before manifest submission.** Collection may already have been transferred to the contract for the next cycle. Since no active manifest exists, none of it is protected as collection.
- **After a manifest is stopped.** The EMT remains in the contract and may be intended for a corrected replacement manifest, but the stopped manifest's `collection_amount` is no longer protected.
- **During batched execution.** Batch 0 applies the waterfall accounting immediately, including deductions from the recorded reserve, while investor transfers can remain pending across later batches. Amounts already assigned to unpaid investors are not included in the sweep earmark.
- **After execution completes.** Any EMT left in the contract is treated as sweepable unless it is still represented by the reserve counter. The contract does not establish that the remaining balance is unrelated to the current or next waterfall cycle before allowing its removal.

For example, assume the contract holds `2` EMT from the current collection and `100` EMT in the recorded reserve. The waterfall uses the full `102` EMT to pay Class A interest, split equally between Alice and Bob. Batch 0 reduces the recorded reserve to zero and pays Alice `51` EMT. Bob's `51` EMT remains in the contract for a later batch.

At that point, `execute_sweep()` protects zero reserve and the original `2` EMT collection amount. It therefore treats `51 - 0 - 2 = 49` EMT as excess. The Service Operator can transfer that amount to the sweep wallet, leaving only `2` EMT and causing Bob's later payment to fail.

The numerical example is one consequence of the broader issue: the contract treats the absence of an internal earmark as proof that EMT is free to remove. No such conclusion can be made from the on-chain information available to `execute_sweep()`.

Impact The Service Operator can remove legitimate Available Amounts from the contract without applying the required priority of payments and without approval from the CoInvestor groups.

Depending on when the sweep occurs, this can:

- remove collection intended for a future or replacement manifest;
- remove EMT allocated to pending investor payments, leaving the current waterfall partially executed and causing later `execute_batch()` calls to fail;
- require the swept funds to be returned or the contract to be funded again before execution can continue.

The sweep destination is fixed in configuration, which prevents the caller from selecting an arbitrary recipient. However, this does not preserve the waterfall's priority of payments or guarantee that the transferred funds will be returned in time.

Recommendation Do not allow `perform_sweep()` to sweep the configured EMT token. The contract cannot reliably distinguish accidentally transferred EMT from legitimate Available Amounts that have not yet been associated with a manifest. Keep the existing sweep operation only for recovering unrelated non-EMT tokens.

```
if *token_addr == config.emt_token {
    return Err(WaterfallError::EmtSweepNotAllowed);
}
```

If the protocol needs a way to recover EMT, implement it as an explicit exceptional governance process instead of deriving a sweepable amount from the token balance.

Developers Response The sweeping mechanism is designed as an emergency function to recover funds that are mistakenly sent to the contract. With that in mind, the earlier restrictions imposed on the sweep mechanism have been removed. Nothing is protected from being swept except for the reserve, which has its own withdrawal function. The earlier restriction that prevented sweeping when the contract was closed has also been removed so that any funds stuck in the contract can still be recovered.

The Service Operator is the administrator responsible for the sweep mechanism and is also responsible for managing the money flows before funds are distributed. Swept funds can only be transferred to the configured `sweep_wallet` and not to an arbitrary on-chain address. Any misuse of these privileges by the Service Operator will be handled through the legal court system.

5.1.7 V-TWFD-VUL-007: Repeated accruals within month count common boundary day twice

Severity	Medium	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/waterfall/src/interest.rs:267-286`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/20		

Description The protocol accrues interest owed to each tranche based on the amount of time that has passed since the previous accrual. Interest is accrued up to a provided target timestamp, with the accrual period being split into segments depending on whether a month boundary is reached. When a segment reaches the next month boundary, the current interest period is closed. Otherwise, the accrual is handled as a partial segment ending at the target timestamp.

For partial segments, the protocol also counts the `target_ts` day towards the interest accrual. To account for this, `accrue_to()` calculates the number of days between the current `cursor` and `target_ts` and adds one additional day to the result. This can be seen in the snippet below.

```
pub fn accrue_to( _env: &Env, tranche: &mut TrancheState, target_ts: u64, compounds: bool, )
{
// veridise: elided

    loop {
    let next_month = calendar::start_of_next_month(cursor);

    if next_month <= target_ts {
        // veridise: elided //
    } else { // Segment ends before the month -> partial segment
        // --- Partial segment ending at target_ts (no period close) ---
        // Payment-day convention: include target_ts day as a full day
        // in this segment (NIA "actual days").  The IPD-split for
        // variable-principal Class A falls out of this naturally.
        let base_days = calendar::days_between(cursor, target_ts);
        let days = if base_days > 0 { base_days + 1 } else { 0 };
        if days > 0 {
            let year_days = calendar::days_in_year(calendar::year_of(cursor));
            let seg_note = calculate_interest(
                tranche.principal_outstanding,
                rate_bps,
                days,
                year_days,
            );
            tranche.interest_outstanding += seg_note;
            tranche.period_accrual += seg_note;
        }
        break;
        }
    }
    tranche.last_accrual_ts = target_ts;
}
```

Snippet 5.3: Snippet from `accrue_to()` in `contracts/waterfall/src/interest.rs`

The problem occurs when multiple partial accruals are performed within the same month. Once a partial accrual is completed, its `target_ts` becomes the starting point for the next accrual. Since the previous segment already included this day as its target date, starting the next partial segment from the same day causes the common boundary day to be counted again.

For example, consider two accruals covering `(15 - 20)` and `(20 - 23)`. The first accrual counts `6` days, including the 20th, while the second accrual counts another `4` days starting from the `20`th. This results in `10` days of interest being accrued. However, accruing the same period through a single `(15 - 23)` segment counts only `9` days. Therefore, splitting the accrual at the 20th results in one additional day of interest being counted.

The implementation already accounts for this shared boundary when accrual continues from a previous partial segment and reaches the next month boundary. In this case, the code recognizes that the previous partial segment already counted the cursor day and starts the next segment from `cursor + SECONDS_PER_DAY`.

```
// When cursor is mid-month (payment split from previous cycle), the
// payment day was counted as a full day in segment 1 via the
// partial-segment +1 convention; this segment starts the day after.
let days = if cursor == calendar::start_of_month(cursor) {
   calendar::days_between(cursor, next_month)
} else {
   let day_after = cursor + calendar::SECONDS_PER_DAY;
   calendar::days_between(day_after, next_month)
};
```

Snippet 5.4: Snippet from `accrue_to()`

However, the same adjustment is not made when one partial segment is followed by another partial segment within the same month. As a result, the common boundary day is counted twice and the total interest accrued depends on how many times the accrual period is split within the month.

Impact Repeated accruals within the same month can cause interest to be overcounted. If accrual within a month is split into multiple segments, each common boundary between segments will result in an additional day of interest being counted.

This will lead to investors receiving more interest than they are owed depending on how often accrual is triggered, even though the actual amount of time that has passed remains the same.

Recommendation When a partial accrual continues from a previous partial segment, start counting from the day after `last_accrual_ts` rather than counting the boundary day again. This should follow the same approach already used when accrual continues from a partial segment into a month boundary, where `SECONDS_PER_DAY` is added to the cursor before calculating the remaining days.

Additionally, add regression tests that ensure accruing a period through multiple calls produces the same total interest as accruing the same period through a single call.

Developer Response The developers updated accrual to avoid counting the same day twice and preserve the accrual timestamp when no days are counted, preventing the first day of the month from being skipped.

5.1.8 V-TWFD-VUL-008: Instance TTL may expire between monthly calls

Severity	Low	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/payout-forwarder/src/storage.rs:19-20`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/12		

Description The payout forwarder extends its instance TTL to only 30 days:

```
pub const INSTANCE_BUMP_AMOUNT: u32 = 30 * DAY_IN_LEDGERS;
pub const INSTANCE_LIFETIME_THRESHOLD: u32 = 29 * DAY_IN_LEDGERS;
```

This period does not safely cover a monthly operating schedule. Two calls made on the same calendar day in consecutive months can be 31 days apart. After the TTL has been reduced to the 30-day range, the contract instance and all data stored in instance storage can therefore be archived before the next scheduled call.

Once the instance is archived, the next call cannot use the contract normally until the archived ledger entries are restored.

Impact Regular monthly operations can be interrupted during 31-day intervals. Forwarding and administrative calls may fail until the contract instance is restored.

Recovery adds operational delay and extra transaction costs. This is especially disruptive if funds are expected to be forwarded on a fixed date.

Recommendation Extend the instance TTL to at least 32 days and keep a one-day renewal margin:

```
pub const INSTANCE_BUMP_AMOUNT: u32 = 32 * DAY_IN_LEDGERS;
pub const INSTANCE_LIFETIME_THRESHOLD: u32 = 31 * DAY_IN_LEDGERS;
```

It is recommended to add a regression test that advances the ledger by 31 days after a TTL extension and verifies that the contract instance remains live and the next call succeeds.

Developer Response The developers have extended the instance TTL to `60` days and the renewal margin to `59` days.

5.1.9 V-TWFD-VUL-009: Sweep can send tokens to their own contract

Severity	Low	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/payout-forwarder/src/lib.rs:259-260`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/13		

Description The `sweep` function prevents the forwarder from sending tokens to itself, but it does not prevent the destination from being the token contract address:

```
let self_addr = env.current_contract_address();
if destination == self_addr {
    return Err(ForwarderError::InvalidAddress);
}

let tok = token::Client::new(&env, &token_addr);
let held = tok.balance(&self_addr);
if held < amount {
    return Err(ForwarderError::InsufficientBalance);
}

tok.transfer(&self_addr, &destination, &amount);
```

An administrator can accidentally call `sweep` with `destination` equal to `token_addr`. The transfer then sends the selected tokens to their own token contract instead of a usable recipient address.

Because the token contract cannot authorize a normal transfer of that balance, the transferred funds can become permanently inaccessible.

Impact All tokens included in the mistaken sweep can be lost. The affected amount is limited only by the forwarder's balance and the amount supplied by the administrator.

The issue requires an authorized administrator to provide the wrong destination, so it is unlikely to happen during normal operation. However, the loss is severe if it does happen.

Recommendation Reject the call when `destination` is either the forwarder address or the selected token contract address:

```
let self_addr = env.current_contract_address();
if destination == self_addr || destination == token_addr {
    return Err(ForwarderError::InvalidAddress);
}
```

It is recommended to add a regression test that calls `sweep` with `destination == token_addr` and verifies that it returns `ForwarderError::InvalidAddress` without changing any token balances.

Developer Response The developers have implemented the recommendation.

5.1.10 V-TWFD-VUL-010: Unprotected initialization allows contract takeover

Severity	Low	**Commit**	809af83
Type	Access Control	**Status**	Fixed
Location(s)	`contracts/` ▶ `payout-forwarder/src/lib.rs:123-124` ▶ `waterfall/src/lib.rs:106-107`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/8		

Description Both the payout forwarder and the Waterfall are deployed first and initialized in a separate call. In both contracts, the first caller can choose the administrator and then satisfy the authorization check using that address.

The payout forwarder uses this pattern:

```
pub fn initialize(
    env: Env,
    admin: Address,
    emt_token: Address,
    xlm_token: Address,
    destination: Address,
    initial_whitelist: Vec<Address>,
    init_locks: InitLocks,
) -> Result<(), ForwarderError> {
    extend_instance_ttl(&env);

    if storage::has_initialized(&env) {
        return Err(ForwarderError::AlreadyInitialized);
    }

    admin.require_auth();

    // ...

    storage::set_admin(&env, &admin);
    storage::set_emt_token(&env, &emt_token);
    storage::set_xlm_token(&env, &xlm_token);
    storage::set_destination(&env, &destination);
    storage::set_whitelist(&env, &deduped);
    storage::set_initialized(&env);

    // ...
    Ok(())
}
```

The Waterfall has the same issue:

```
pub fn initialize(
    env: Env,
    admin: Address,
    config: Configuration,
    initial_tranches: AllTrancheState,
```

```
    class_a_holders: Vec<Address>,
    class_b_holders: Vec<Address>,
    class_c_holders: Vec<Address>,
    holder_payout_overrides: Vec<(Address, Address)>,
) -> Result<(), WaterfallError> {
    extend_instance_ttl(&env);

    if storage::has_initialized(&env) {
        return Err(WaterfallError::AlreadyInitialized);
    }

    admin.require_auth();

    // ...
}
```

In both cases, `admin.require_auth()` does not protect initialization because the caller supplies `admin`. An attacker can provide their own address, sign the call, and take over any deployed but uninitialized instance.

For the payout forwarder, the attacker can set their own destination and whitelist and enable the initialization locks. For the Waterfall, the attacker can choose the `permission55` contract, token addresses, outgoing wallets, tranche state, holder lists, and payout overrides. A malicious `permission55` configuration can then give the attacker the roles required to operate the Waterfall.

Initialization is one-shot in both contracts. After the attacker completes it, the intended deployer cannot repair the instance through another `initialize` call.

Impact If an uninitialized forwarder is connected to the Waterfall or funded without checking its configuration, an attacker can control forwarded funds or sweep the contract balance.

If a compromised Waterfall address is funded or used without checking its configuration, the attacker can route funds through attacker-controlled outgoing wallets or use roles provided by the malicious `permission55` contract. This can lead to the loss of all funds sent to either compromised contract.

If the deployment process verifies each contract's configuration before using it, the takeover should be detected and the affected instance can be abandoned and redeployed. Exploitation therefore depends on the gap between deployment and initialization and a missing operational check, making it unlikely.

Recommendation Move the critical configuration of both contracts into their constructors so deployment and initialization happen atomically. Each constructor should require authorization from the intended administrator, validate the complete configuration, and store all initial state before deployment completes.

For example:

```
pub fn __constructor(
    env: Env,
    admin: Address,
    emt_token: Address,
    xlm_token: Address,
```

```
    destination: Address,
    initial_whitelist: Vec<Address>,
    init_locks: InitLocks,
) -> Result<(), ForwarderError> {
    admin.require_auth();

    validate_initial_configuration(
        &env,
        &admin,
        &emt_token,
        &xlm_token,
        &destination,
        &initial_whitelist,
    )?;

    store_initial_configuration(
        &env,
        &admin,
        &emt_token,
        &xlm_token,
        &destination,
        &initial_whitelist,
        &init_locks,
    );

    Ok(())
}
```

Apply the same pattern to the Waterfall constructor, including its `Configuration`, initial tranche state, holder lists, and payout overrides. Do not rely on a role check through `permission55` to protect initialization because the current caller can choose the `permission55` address.

Emit a `Deployed(admin)` event from each constructor. The off-chain deployment service should compare the event's contract address and arguments with the expected values before publishing or funding the contract. This check provides an additional way to detect deployment mistakes and does not replace atomic constructor-based initialization.

Remove the public one-shot initialization paths after the constructors are introduced.

Developer Response The developers have implemented the recommendation.

5.1.11 V-TWFD-VUL-011: Service Operator with coinvestor role can stop manifest after approval deadline

Severity	Low	**Commit**	809af83
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/waterfall/src/manifest.rs:232-244`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/22		

Description The Service Operator and CoInvestors have different restrictions on when they are allowed to stop an ongoing manifest. In particular, a CoInvestor can no longer stop the manifest once their respective approver set has reached quorum. Similarly, the Service Operator loses the authority to stop the manifest once the approval deadline has passed.

However, these restrictions do not work as intended when the Service Operator is also a CoInvestor. As shown below, the CoInvestor quorum restriction is only applied when the caller does not also have the Service Operator role through the `!has_so` check. Similarly, the Service Operator's approval deadline is only enforced when the caller does not hold either CoInvestor role through the `!has_a && !has_b` checks.

```
pub fn stop(env: &Env, caller: &Address) -> Result<(), WaterfallError> {
  let lifecycle = storage::get_lifecycle(env);
  state_machine::require_active(&lifecycle)?;
  // veridise: elided //

  // F-2: a CoInvestor cannot stop once their own side is locked in.
  let a_full = ws.co_investor_a_approvers.len() >= 2;
  let b_full = ws.co_investor_b_approvers.len() >= 2;
  if has_a && a_full && !has_so {
      return Err(WaterfallError::QuorumAlreadyReached);
  }
  if has_b && b_full && !has_so {
      return Err(WaterfallError::QuorumAlreadyReached);
  }
  // F-3: SO loses stop authority after the approval deadline; the only
  // post-deadline forward path is the timeout fallback in execute_batch.
  if has_so && !has_a && !has_b && env.ledger().timestamp() > ws.approval_deadline {
      return Err(WaterfallError::ApprovalWindowClosed);
  }
}
```

Snippet 5.5: Snippet from `stop()` in `contracts/waterfall/src/manifest.rs`

Therefore, if the Service Operator also holds a CoInvestor role, neither restriction applies as expected. The Service Operator can still stop the manifest even after their respective CoInvestor approver set has reached quorum. And more importantly, they can stop the manifest even after the approval deadline has passed, violating the intended invariant that the Service Operator loses its stop authority after the approval deadline.

Impact The Service Operator retains the ability to stop the manifest beyond the intended approval deadline if they also hold a CoInvestor role. This allows the Service Operator to block

the manifest at a point where they should no longer have the authority to do so, weakening the intended governance restrictions around the waterfall execution.

Recommendation Clarify the intended behavior for manifests that have passed the approval deadline and align the implementation and relevant documentation accordingly.

If the Service Operator is not intended to stop a manifest after the approval deadline, prevent `stop_manifest()` from being called once the deadline has passed. Otherwise, if this behavior is intentional, clearly document that a Service Operator with the coinvestor role retains the ability to stop the manifest after the approval deadline.

Alternatively, if the intention was to assign mutually exclusive roles to participants, enforce this behavior so that no single address can hold two roles at once.

Developer Response The developers now prevent all callers, including the Service Operator, from stopping an approved manifest.

5.1.12 V-TWFD-VUL-012: Service Operator can close waterfall before all investor payments executed

Severity	Low	**Commit**	809af83
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/waterfall/src/lib.rs:940-975`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/14		

Description The Service Operator can close the waterfall once the deal has been settled by calling `close()`, which transitions the waterfall into the terminal `Closed` state. To prevent the waterfall from being closed prematurely, this is only allowed when Class A and Class B have no remaining `payable_outstanding()`, Class C has been reduced to its residual amount, and the reserve balance is empty.

However, these checks only consider whether the tranche accounting has been settled and do not verify that the corresponding payments to investors have actually been completed. This is problematic because waterfall distributions may be processed over multiple batches. The accounting for the distribution will already be updated during the initial batch, while payments to the remaining investors may still be pending. At that point, the requirements for `close()` will already be satisfied even though the execution plan has not yet finished paying every holder.

```rust
pub fn close(env: Env, caller: Address) -> Result<(), WaterfallError> {
    //veridise: elided //

    let tranches = storage::get_tranche_state(&env);
    let config = storage::get_config(&env);
    let residual = waterfall::class_c_residual(config.emt_decimals);

    if tranches.class_a.payable_outstanding() != 0
        || tranches.class_b.payable_outstanding() != 0
        || tranches.class_c.payable_outstanding() > residual
    {
        return Err(WaterfallError::TranchesNotSettled);
    }

    if storage::get_reserve_balance(&env) != 0 {
        return Err(WaterfallError::ReserveNotEmpty);
    }

    let released = Self::release_class_c_residual(&env)?;
    storage::set_lifecycle(&env, &LifecycleState::Closed);

    //veridise: elided //
}
```

Snippet 5.6: Snippet from `close()` in `contracts/waterfall/src/lib.rs`.

The `Closed` state is terminal and there is no transition out of that state, as can be seen below in the lifecycle state transition rules.

```rust
pub fn is_valid_transition(from: &LifecycleState, to: &LifecycleState) -> bool {
   matches!(
```

```
        (from, to),
        (LifecycleState::Setup, LifecycleState::Active)
            | (LifecycleState::Active, LifecycleState::Paused)
            | (LifecycleState::Paused, LifecycleState::Active)
            | (LifecycleState::Active, LifecycleState::Closed)
            | (LifecycleState::Paused, LifecycleState::Closed)
    )
}
```

Snippet 5.7: Snippet from `is_valid_transition()` in `contracts/waterfall/src/lib.rs`

As a result, if the Service Operator calls `close()` after the accounting for the final distribution has been settled, but before all payout batches have completed, the lifecycle is moved to `Closed`. The remaining investor payments can then no longer be processed, leaving the corresponding funds stranded in the contract with no normal recovery path.

The Service Operator is a trusted entity and would not be expected to intentionally close the waterfall while a payout is still in progress. However, the current checks permit this state to be reached through an operational mistake, and there is no recovery once the contract has entered the terminal `Closed` state.

Impact The Service Operator can accidentally or intentionally close the waterfall while investor payments are still pending. Because `Closed` is terminal, remaining investor funds will become permanently stranded in the contract.

Recommendation Prevent `close()` from succeeding while a waterfall execution plan still has pending investor payments. The contract should require both the tranche accounting and the corresponding payment execution to be fully settled before transitioning to `Closed`.

Developer Response The developers added a check that rejects `close()` while an execution plan exists, preventing closure before all payment batches complete.

5.1.13 V-TWFD-VUL-013: Pending approvals can apply to different manifest

Severity	Low	**Commit**	809af83
Type	Authorization	**Status**	Fixed
Location(s)	`contracts/waterfall/src/lib.rs:366-367, 379-380`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/15		

Description The authorization for approving or stopping a manifest is not linked to the manifest that the caller reviewed. Both public functions accept only the caller address:

```
pub fn approve(env: Env, caller: Address) -> Result<ApprovalStatus, WaterfallError> {
    extend_instance_ttl(&env);
    manifest::approve(&env, &caller)
}

pub fn stop_manifest(env: Env, caller: Address) -> Result<(), WaterfallError> {
    extend_instance_ttl(&env);
    manifest::stop(&env, &caller)
}
```

`caller.require_auth()` proves that the caller authorized an `approve(caller)` or `stop_manifest(caller)` invocation. It does not prove which active manifest the caller intended to approve or stop.

For example:

1. The ServiceOperator submits manifest M1.
2. An investor reviews M1 and signs an approval, but the transaction is delayed for some reason.
3. M1 is stopped and the ServiceOperator submits M2.
4. The delayed transaction executes and records the investor's approval for M2, although the investor never reviewed M2.

The same problem affects `stop_manifest`. A delayed stop signed for M1 can stop M2 if another authorized participant stops M1 and M2 is submitted before the delayed transaction executes.

Binding the calls only to `cycle_id` would not fix the problem. A stopped manifest does not complete the cycle, so its replacement uses the same `cycle_id`.

Impact An investor's valid authorization can be applied to a different manifest from the one they reviewed. A replacement manifest can therefore receive approvals that were intended for an earlier submission.

If this helps the replacement reach quorum, the Waterfall can execute with unapproved collection amounts and distribute funds using parameters that the affected investors did not accept. A delayed stop can also cancel a valid replacement manifest and delay the payment cycle.

The attack requires manifest replacement while signed transactions are still pending, and normally requires a malicious or compromised ServiceOperator or transaction relayer.

Recommendation Assign a unique, monotonically increasing `submission_id` to every submitted manifest, including replacements within the same cycle. Store this ID in `WaterfallState` and increment it on every successful `submit_manifest` call.

Derive a commitment from the active manifest fields when it is needed. The commitment should cover the `cycle_id`, `submission_id`, `manifest_hash`, `collection_amount`, `amount_after_expenses`, `servicer_balance`, and `approval_deadline`.

Change both `approve` and `stop_manifest` so their signed arguments include only the expected manifest commitment:

```
pub fn approve(
    env: Env,
    caller: Address,
    manifest_commitment: BytesN<32>,
) -> Result<ApprovalStatus, WaterfallError> {
    extend_instance_ttl(&env);
    manifest::approve(&env, &caller, &manifest_commitment)
}

pub fn stop_manifest(
    env: Env,
    caller: Address,
    manifest_commitment: BytesN<32>,
) -> Result<(), WaterfallError> {
    extend_instance_ttl(&env);
    manifest::stop(&env, &caller, &manifest_commitment)
}
```

Before recording an approval or stopping a manifest, compute the commitment from the active `WaterfallState` and compare it with the supplied value. Reject the call if they do not match. Because the commitment is part of the signed function arguments, an authorization prepared for M1 cannot be applied to M2. The unique `submission_id` ensures that the commitments differ even when both submissions have otherwise identical payloads and share the same `cycle_id`.

It is recommended to add regression tests for both paths. Each test should authorize an action for M1, stop and replace M1 with M2 in the same cycle, then verify that the old approval or stop authorization is rejected and does not modify M2.

Developers Response The developers implemented a monotonically increasing `submission_id`, which is tracked with each manifest submission. Approval and stoppage of a manifest now requires passing in the expected commitment hash.

5.1.14 V-TWFD-VUL-014: One-step admin rotation can permanently lose control

Severity	Low	**Commit**	809af83
Type	Access Control	**Status**	Fixed
Location(s)	`contracts/` ▶ `payout-forwarder/src/lib.rs:350-351` ▶ `permission55/src/lib.rs:50-56`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/16		

Description Both the payout forwarder and Permission55 change their administrator in a single transaction. The current administrator authorizes the call, but each contract immediately replaces their address with `new_admin`.

The payout forwarder uses the following logic:

```
pub fn set_admin(env: Env, admin: Address, new_admin: Address) -> Result<(),
    ForwarderError> {
    extend_instance_ttl(&env);
    require_admin(&env, &admin)?;
    if storage::get_admin_locked(&env) {
        return Err(ForwarderError::AdminLocked);
    }

    let emt = storage::get_emt_token(&env);
    let xlm = storage::get_xlm_token(&env);
    validate_routable_address(&env, &new_admin, &emt, &xlm)?;

    storage::set_admin(&env, &new_admin);
    events::admin_changed(&env, &admin, &new_admin);
    Ok(())
}
```

Permission55 follows the same one-step pattern:

```
pub fn set_admin(e: Env, new_admin: Address) -> Result<(), Error> {
    let admin = read_administrator(&e)?;
    admin.require_auth();
    bump_instance(&e);

    write_administrator(&e, &new_admin);
    events::NewAdmin { new_admin: new_admin.clone() }.publish(&e);
    Ok(())
}
```

The payout forwarder's address validation rejects known contract addresses, but it cannot prove that anyone controls `new_admin`. Permission55 does not require the new address to prove control either. If the current administrator enters the wrong address or an address whose key is unavailable, the rotation still succeeds in either contract.

The payout forwarder's code comments state that a mistaken rotation can be repaired by the old administrator if it is discovered before the old key is lost. This is incorrect. As soon as `storage::set_admin` completes, the old address is no longer the administrator. Permission55

has the same result after `write_administrator` completes. In either contract, the old address cannot repair the rotation even if its key is still available.

Impact A single address mistake can permanently remove administrative control over either contract. In the payout forwarder, the administrator can no longer change the destination or whitelist, rotate the administrator, apply locks, or recover tokens through `sweep`.

In Permission55, the lost administrator can no longer rotate the administrator, manage minter and burner privileges, create or remove permission sets, or update token URIs. This can prevent the protocol from changing the roles used by contracts that rely on Permission55.

If EMT or other tokens are held by the contract when this happens, recovery may require deploying a replacement forwarder and updating every system that sends funds to it. Tokens already held by the original contract may remain inaccessible.

The issue requires an authorized administrator to submit an incorrect or inaccessible address, so it is unlikely during normal operation. However, the consequences can be severe and irreversible.

Recommendation Use a two-step administrator transfer in both the payout forwarder and Permission55:

1. The current administrator calls `propose_admin(new_admin)`. The contract stores `new_admin` as the pending administrator but does not change the active administrator.
2. The pending administrator calls `accept_admin()` and authorizes the call. Only then does the contract replace the active administrator and clear the pending value.

The active administrator should retain all existing permissions until the pending administrator accepts. They should also be able to cancel the pending transfer or replace it with a different address. Emit an event when a transfer is proposed or cancelled, and emit the final administrator-change event only after acceptance.

For the payout forwarder, define how `lock_admin` interacts with a pending transfer. A simple safe rule is to reject `lock_admin` while a transfer is pending, requiring the administrator to complete or cancel the transfer first.

Add regression tests for both contracts that verify:

- proposing a transfer does not remove the current administrator's permissions;
- only the pending address can accept the transfer;
- the old administrator loses access only after acceptance; and
- the current administrator can cancel or replace a pending transfer.

For the payout forwarder, also verify that `lock_admin` cannot leave a pending transfer in an ambiguous state.

Developers Response The developers implemented a two step administrator transfer for both the payout forwarder and Permission55.

5.1.15 V-TWFD-VUL-015: Duplicate tranche holders can distort pro rata distributions

Severity	Low	**Commit**	809af83
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/waterfall/src/lib.rs:315-320`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/17		

Description The holder registry for each tranche is provided during initialization and is used to determine which investors participate in the corresponding tranche distributions. The same investor may validly appear across different tranches, since an investor can hold positions in more than one tranche. However, each individual tranche holder list is expected to contain a unique set of investors.

Currently, `initialize()` only checks that the number of holders in each tranche remains below `MAX_HOLDERS_PER_CLASS`. It does not verify that the holder lists contain unique addresses before they are stored in the `HolderRegistry`.

```
  pub fn initialize(env: Env,
    admin: Address,
    config: Configuration,
    initial_tranches: AllTrancheState,
    class_a_holders: Vec<Address>,
      class_b_holders: Vec<Address>,
      class_c_holders: Vec<Address>,
      holder_payout_overrides: Vec<(Address, Address)>,
   ) -> Result<(), WaterfallError> {
         if class_a_holders.len() > MAX_HOLDERS_PER_CLASS
            || class_b_holders.len() > MAX_HOLDERS_PER_CLASS
            || class_c_holders.len() > MAX_HOLDERS_PER_CLASS
         {
            return Err(WaterfallError::TooManyHolders);
         }

         // veridise: elided //

         let registry = HolderRegistry {
            class_a_holders,
            class_b_holders,
            class_c_holders,
         };
         storage::set_holder_registry(&env, &registry);

     // veridise: elided //
}
```

Snippet 5.8: Snippet from `initialize()` in `contracts/waterfall/src/lib.rs`

As a result, the same holder can currently be included multiple times within a single tranche. This breaks the assumptions made by the pro rata distribution logic, since the holder's token balance will be considered multiple times when iterating through the holder list. The duplicated

holder may therefore receive a larger portion of the tranche distribution while reducing the amount available to the other investors.

For example, if the Class A registry is initialized as `[Alice, Alice, Bob]`, Alice will be treated as two separate entries even though both entries refer to the same token balance. Her balance can therefore contribute multiple times to the distribution calculations and she may also be processed multiple times during investor payout execution.

This is particularly problematic because the holder registry is immutable after initialization, meaning that an incorrectly initialized registry cannot be corrected through normal means once the contract becomes active.

Impact Duplicate addresses within a tranche holder list can distort pro rata distribution accounting and cause investors to receive incorrect payout amounts. Moreover, since the registry is immutable after initialization, an incorrectly seeded registry is not something that can be readily fixed afterwards.

Recommendation

1. Reject duplicate addresses during initialization.
2. Check each tranche list separately, because the same investor may validly hold tokens in different tranches.
3. Add a dedicated `DuplicateHolder` error so deployment tooling can clearly explain why initialization failed.

Developer Response The developers make use of `has_duplicate_holder()` during initialization to ensure there are no duplicate holders within a tranche.

5.1.16 V-TWFD-VUL-016: Code quality improvements

Severity	Warning	**Commit**	809af83
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/` ▶ `payout-forwarder/src/lib.rs:156-165, 175` ▶ `waterfall/src/` • `errors.rs:196` • `lib.rs:106-334, 242, 290-298, 600-609` • `manifest.rs:92-98` • `types.rs:58`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/25		

Description Several code quality concerns were identified throughout the codebase. These include unused code, events that do not expose all relevant information, and areas where additional defensive checks could improve the robustness of the implementation. The identified instances are outlined below.

Unused Code:

- The event `EventOfDefault` in `contracts\waterfall\src\errors.rs` is unused.
- The field `final_maturity_date` within the `Configuration` struct in `contracts\waterfall\src\types.rs` is not used anywhere. This is a problem because the contract might continue accruing interest past the final maturity date. If this field is a relic of development then it should be removed, and if otherwise the contract should validate that past the final maturity date no accrual of interest occurs. Moreover, it also makes sense to not allow `close()` before the final maturity date because the debtor's portfolio has not matured yet and there is still interest to be accrued.

Missing fields in events:

- Inside `contracts\waterfall\src\manifest.rs` the `submit()` function does not publish `amount_after_expenses` as well as `servicer_balance` when `events::manifest_submitted()` is submitted. This is concerning because `get_manifest()` returns only the current manifest, and once a new manifest replaces it the current manifest will be replaced. Afterwards the historical manifest execution terms for the past manifests cannot be reconstructed purely from on-chain data or event tracking.
- Inside `contracts/payout-forwarder/src/lib.rs`, the `initialize()` function does not emit the initial whitelist members as part of `events::initialized()`. This can make recovering the original whitelist content harder when it is needed. Moreover, the emitted event does not contain the `init_locks` which are set just after the event is emitted. Both of these fields provide vital information that should be emitted as part of the `events::initialized()` event.
- In `execute_batch()` inside `contracts/waterfall/src/lib.rs`, when the `waterfall_completed()` event is emitted, it uses `amount_distributed` in place of the `total_distributed` amount that is mentioned in the event definition. The total distribution amount is the correct amount to emit here as the `waterfall_completed()` event is triggered when the waterfall distributes its final batch of the execution cycle.

Defense in Depth recommendations:

- In `contracts/payout-forwarder/src/lib.rs`, when the whitelist is checked for duplicates in `initialize()`, it is recommended to explicitly enforce the maximal length of the whitelist to exclude potential out-of-gas DoS scenarios. If `require_whitelisted()` can't process the full list, this state may no longer be recoverable: the `remove_caller()` call will abort as well since it traverses and rebuilds the list.
- In `contracts/waterfall/src/lib.rs`, the `initialize()` function permits specifying empty holder lists for each tranche, despite requiring each tranche to have a positive initial principal. This may result in a tranche containing value to be distributed without any registered holders to receive the corresponding payments. Consider requiring each tranche to have at least one holder during initialization.
- In `contracts/waterfall/src/lib.rs`, `initialize()` accepts any nonzero `last_accrual_ts` supplied for the tranches and only replaces the value with the current ledger timestamp when it is `0`. This can be problematic because timestamps dated in the future may be accepted, in which case no interest will accrue until that date. Require any provided `last_accrual_ts` to not be greater than the current ledger timestamp.
- In `contracts/waterfall/src/lib.rs`, the fresh initialization path only requires `interest_outstanding == 0` when `last_accrual_ts == 0`. It does not require `completed_period_interest`, `period_accrual`, or `default_interest_outstanding` to also be zero. This can allow the waterfall to initialize with inconsistent interest accounting state. Require all carried interest fields to be zero for a fresh initialization.

Impact The identified concerns may make the codebase harder to maintain, reduce the usefulness of emitted events for off-chain monitoring, and introduce unnecessary operational risks in edge cases.

Recommendation Address the identified code quality concerns by removing or properly integrating unused code, ensuring events expose the relevant state changes and information, and adding the suggested defensive checks wherever appropriate.

Developers Response The developers have addressed the code quality concerns, as recommended.

5.1.17 V-TWFD-VUL-017: Token addresses can point to payout forwarder

Severity	Warning	**Commit**	809af83
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/payout-forwarder/src/lib.rs:148-149`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/18		

Description The payout forwarder's initialization comments state that `emt_token`, `xlm_token`, and `destination` must be different from the contract's own address. The implementation only checks that the two token addresses differ from each other:

```
1 if emt_token == xlm_token {
2     return Err(ForwarderError::InvalidAddress);
3 }
4
5 validate_routable_address(&env, &destination, &emt_token, &xlm_token)?;
6 validate_routable_address(&env, &admin, &emt_token, &xlm_token)?;
```

`validate_routable_address` prevents `destination` and `admin` from pointing to the forwarder or either configured token contract. It does not validate `emt_token` or `xlm_token` themselves. Initialization therefore succeeds if either token address equals `env.current_contract_address()`.

If `emt_token` points to the payout forwarder, `forward` creates a token client for the payout forwarder and tries to call the token `balance` and `transfer` functions on it. The payout forwarder does not implement those functions, so normal EMT forwarding fails.

If `xlm_token` points to the payout forwarder, `send_xlm` similarly tries to call a token `transfer` function on the payout forwarder and fails.

The configured token addresses cannot be changed after initialization, so the affected function remains unusable for that deployment.

Impact An incorrect token address can disable all normal EMT forwarding or XLM distribution through the deployed contract. The deployment must be replaced or downstream integrations must be reconfigured before normal operation can continue.

EMT already held by the payout forwarder may still be recoverable by an active administrator using `sweep` with the real EMT token address. However, recovery and redeployment cause operational delays and additional transaction costs.

The issue requires an incorrect address during deployment, so it is unlikely during normal operation.

Recommendation Reject either configured token address if it equals the payout forwarder's own address:

```
let self_addr = env.current_contract_address();
if emt_token == xlm_token || emt_token == self_addr || xlm_token == self_addr {
    return Err(ForwarderError::InvalidAddress);
}
```

Keep the existing validation for `admin` and `destination`.

Add regression tests that initialize the contract with `emt_token` equal to the forwarder address and with `xlm_token` equal to the forwarder address. Both calls should return `ForwarderError::InvalidAddress` and leave the contract uninitialized.

Developers Response The developers have implemented the recommended check.

5.1.18 V-TWFD-VUL-018: XLM relay can use arbitrary token

Severity	Warning	**Commit**	809af83
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/waterfall/src/lib.rs:148-151, 946-950`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/19		

Description The Waterfall specification describes `Configuration.xlm_token` as the native XLM SAC address and states that storing this address during initialization locks `send_xlm` to XLM. The implementation does not verify that the supplied address is the canonical native XLM SAC for the current network.

Initialization only checks that `xlm_token` differs from EMT and the three tranche-token addresses:

```
if config.xlm_token == config.emt_token
    || config.xlm_token == config.class_a_token
    || config.xlm_token == config.class_b_token
    || config.xlm_token == config.class_c_token
{
    return Err(WaterfallError::InvalidAddress);
}
```

`send_xlm` later creates a generic token client at the stored address and reports each successful transfer as XLM:

```
let config = storage::get_config(&env);
let xlm = token::Client::new(&env, &config.xlm_token);

for i in 0..num_destinations {
    let dest = destinations.get(i).unwrap();
    xlm.transfer(&caller, &dest, &amount_per_destination);
    events::xlm_sent(&env, &caller, &dest, amount_per_destination);
}
```

Any other compatible SEP-41 token contract can therefore be configured in the XLM slot. Calls to `send_xlm` would transfer that token while emitting `XlmSent` events. If the configured address does not implement the expected token interface, the call fails instead.

The official CLI derives the native XLM SAC from the network passphrase by default, which reduces the chance of accidental misconfiguration. However, the CLI also permits an explicit `--xlm-token` override, and neither behavior creates an on-chain guarantee.

The legal Waterfall requirements do not define the XLM relay. It is an operational utility added to fund investor accounts before EMT-provider whitelisting, rather than part of the core payment waterfall.

Impact A deployment configured with the wrong token cannot use `send_xlm` to provide investors with the native XLM required for account funding, transaction fees, or the intended EMT-whitelisting process. Operations may fail until the Waterfall is redeployed with the correct immutable configuration.

If the configured address points to another compatible token, a ServiceOperator can transfer that token from their own address to the supplied destinations, and the resulting events incorrectly identify the transfers as XLM.

Recommendation If the contract is intended to guarantee XLM-only behavior, compare `config.xlm_token` with the canonical native XLM SAC address for the active network during initialization and return `WaterfallError::InvalidAddress` on a mismatch.

The expected address must come from a source that the initializer cannot replace together with the configuration. Depending on the deployment design, this can be implemented by deriving the native SAC address from the network identifier or by using a network-specific value fixed by the contract or its trusted deployment process.

Developers Response The developers now pin the `xlm_token` to the network's canonical native SAC during initialization.

5.1.19 V-TWFD-VUL-019: Interest rounding does not account for configured EMT decimals

Severity	Warning	**Commit**	809af83
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/waterfall/src/` ▶ `interest.rs:58` ▶ `lib.rs:194-197`		
Confirmed Fix At	https://gitlab.com/tokenforge-gmbh/contracts/stellar/eos/-/merge_requests/23		

Description The EMT token used by the Waterfall is configurable through `config.emt_token`, while its decimals are configured separately through `config.emt_decimals`. The implementation accepts `emt_decimals` values between `1` and `18`, but does not verify that the supplied value matches the decimals of the configured EMT token.

```
// EMT decimals must be in a sane range (1..=18)
if config.emt_decimals == 0 || config.emt_decimals > 18 {
   return Err(WaterfallError::InvalidConfiguration);
}
```

Snippet 5.9: Snippet from `initialize()` in `contracts/waterfall/src/lib.rs`

Despite supporting different decimal configurations, the interest rounding logic assumes that EMT always has `6` decimals. In particular, `floor_to_cent()` hardcodes `10,000` base units as the value of one cent:

```
const STROOPS_PER_CENT: i128 = 10_000;

fn floor_to_cent(stroops: i128) -> i128 {
   (stroops / STROOPS_PER_CENT) * STROOPS_PER_CENT
}
```

Snippet 5.10: Snippet from interest.rs

This is correct for a token with `6` decimals, where `0.01` EMT corresponds to `10,000` base units. However, for a token with `d` decimals, one cent corresponds to `10^(d - 2)` base units.

For example, with a `4` decimal EMT token, one cent corresponds to `100` base units, while the current implementation rounds in increments of `10,000` base units, or `1` EMT. An accrued interest amount of `1.99` EMT would therefore be rounded down to `1` EMT, even though it is already an exact number of cents. Conversely, with `7` decimals, one cent corresponds to `100,000` base units, causing the current implementation to retain sub-cent amounts.

`floor_to_cent()` is used when closing periods for normal interest and, for Classes A and B, default interest. Incorrectly rounded interest can therefore also propagate into subsequent default interest calculations. The independently configured `emt_decimals` value is also used elsewhere. For example, the Class C residual is calculated using `10^emt_decimals`. If the actual EMT token has `6` decimals but the protocol is configured with `emt_decimals = 7`, the amount intended to represent `1` EMT instead corresponds to `10` EMT.

The current CLI configuration uses `6` decimals, for which the rounding constant is correct. However, the contract explicitly accepts decimal configurations from `1` to `18` and does not verify that the configured value matches the actual EMT token.

Impact For EMT tokens that do not use `6` decimals, interest can be rounded incorrectly, causing investors to receive more or less interest than intended. This can also carry over into later default interest calculations.

Additionally, if the configured `emt_decimals` does not match the actual EMT token, other calculations that rely on this value, such as the Class C residual, can use an incorrect EMT amount.

Recommendation Read the EMT token's actual decimals during initialization rather than relying solely on a caller-supplied value, and use the resulting decimal value consistently across all decimal-dependent calculations.

For interest rounding, derive the number of base units representing one cent from `emt_decimals`, such as `10_i128.pow(emt_decimals - 2)` for tokens with at least two decimals. Also, explicitly define the expected behavior for tokens with fewer than two decimals if they are intended to be supported.

Developer Response The developers now validate EMT decimals against the configured `emt_token` address, and calculate interest-rounding precision using the validated decimal count.

A Appendix

A.1 Intended Behavior: Non-Issues of Note

A.1.1 V-TWFD-APP-VUL-001: Default interest charges a full month instead of the overdue period

Severity	Medium	**Commit**	809af83
Type	Logic Error	**Status**	Intended Behavior
Location(s)	`contracts/waterfall/src/` ▶ `interest.rs:233` ▶ `waterfall.rs:556`		

Description Interest accrued during a particular month becomes due on the Interest Payment Date of the following month, which is generally the 15th, subject to business-day adjustment. The due date for each completed interest period is calculated and stored in an `OverdueBucket` together with the corresponding amount.

```
/// Interest accrued during [01.MM, 01.(MM+1)) is due on the
/// Interest Payment Date of the following month.
pub fn due_date_for_period(period_start_ts: u64) -> u64 {
    let next_month_start = start_of_next_month(period_start_ts);
    let (y, m, _) = timestamp_to_ymd(next_month_start);
    interest_payment_date(y, m)
}
```

Snippet A.1: Snippet of `due_date_for_period()` from `calendar.rs`

However, these due dates are not taken into account when default interest is calculated. Instead, `accrue_to()` calculates default interest when crossing a calendar-month boundary using the unpaid interest carried over from prior periods and the full number of days in the month.

```
let period_start = calendar::start_of_month(cursor);
let days_in_month = calendar::days_between(period_start, next_month);

let prior_cumulative = tranche
    .completed_period_interest
    .saturating_add(tranche.default_interest_outstanding);

let period_default = if compounds {
    let raw = calculate_interest(
        prior_cumulative,
        default_rate_bps,
        days_in_month,
        year_days,
    );
    floor_to_cent(raw)
} else {
```


© 2026 Veridise Inc.

```
    0
};
```

Snippet A.2: Snippet from `accrue_to()` in `interest.rs`

This means that default interest is calculated based on whether the interest remains unpaid at the end of the month rather than the amount of time for which it was actually overdue.

For example, consider interest accrued during September. This interest becomes due on October 15th, assuming no business-day adjustment. If the amount remains unpaid when accrual crosses into November, the implementation calculates default interest using all `31` days of October. This includes October 1st through 14th, even though the payment was not yet due during this period.

The opposite behavior can occur if the outstanding interest is paid after its due date but before the end of the month. For example, if the same September interest is paid on October 20th, `accrue_to()` does not calculate default interest for the October 15th to 20th overdue period because partial-month accrual only calculates normal interest. The subsequent payment then reduces `completed_period_interest`, removing the paid amount from the base that would otherwise be used to calculate default interest when the month eventually closes.

As a result, an amount that remains unpaid through the end of the month can be charged default interest for days before it was due, while an amount paid after its due date but before month-end can avoid default interest for the period during which it was actually overdue. Partial payments similarly cause only the balance remaining at the end of the month to be charged, with that remaining balance being charged for the full month rather than based on how long each amount was overdue. Therefore, the default interest charged depends on whether an outstanding balance survives until the calendar month close rather than the actual period for which the balance remained overdue.

Impact If default interest is intended to apply only after an interest payment becomes due, the current calculation can both overcharge and undercharge default interest. Amounts remaining unpaid at month-end may be charged for periods before their due date, while amounts paid after their due date but before month-end may avoid default interest entirely.

This can cause the interest owed to Classes A and B to differ from the amount implied by the actual period for which their payments remained overdue.

Recommendation Confirm with the relevant legal and operational stakeholders whether Default Interest must follow the actual overdue period or the existing full-calendar-month convention.

If Default Interest must accrue only from the applicable payment due date until payment, update the calculation to use the due dates stored in the overdue buckets. Accrue Default Interest before applying any full or partial interest payment so that the paid amount is charged only for the time during which it was overdue.

If the full-calendar-month convention is the intended and formally accepted approach, document that due dates do not control Default Interest accrual and that intra-month payments do not produce a separate partial-month Default Interest charge.

The legal basis, accounting documentation, and contract comments should all describe the same rule.

Developer Response The customer has decided that he agrees with us that we go with how their Excel did it and how it is currently practiced. And that is exactly the full month view that we have implemented. This also means it won't trigger for A in reality at all, since, in the current state of the contract, if the payment is delayed by more than 5 business days, it goes to the off-chain default process. So, the current implementation is intended behavior.