Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Centiiv

Centiiv 2.0: Liquidity Infrastructure

Veridise Inc.

April 13, 2026

► **Prepared For:**

Centiiv
www.centiiv.io

► **Prepared By:**

Aayushman Thapa Magar
Evgeniy Shishkin

► **Contact Us:**

contact@veridise.com

► **Version History:**

Apr. 22, 2026	V1
Apr. 21, 2026	Initial Draft

Contents

1 Executive Summary

From Apr. 13, 2026 to Apr. 15, 2026, Centiiv engaged Veridise to conduct a security assessment of the Centiiv Protocol. Veridise conducted the assessment over 6 person-days, with 2 security analysts reviewing the project over 3 days on commit `d758437`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts, as well as thorough code review.

Project Summary. Centiiv is the on-chain part of a USDC-to-cash workflow. A user sends USDC into a fresh temporary wallet generated by the Operator, and the protocol then records whether the order is settled as cash is delivered off-chain or refunded if the Operator cannot fill the remaining amount.

The protocol interacts with the following abstract actors:

- **User**: wants to exchange USDC for cash and, if the order cannot be filled, recover the remaining USDC to their refund address.
- **Operator**: interacts with the User off-chain, controls the relayer and the temporary wallet, delivers cash, settles filled portions on-chain, or refunds the remainder when the order cannot be completed.
- **Liquidity Provider Contract**: records order state on-chain, tracks settlement progress, and executes the USDC transfers that correspond to settlement fees, operator payout, or User refund.

The main workflows are:

1. **Order creation and funding**: the Operator provides a fresh temporary wallet, an order is created for the User, and USDC is transferred from the User's wallet to that temporary wallet.
2. **Settlement**: when the Operator delivers cash, they call `settle()` with the filled percentage, then execute the settlement transfer so the corresponding USDC is released from the temporary wallet.
3. **Refund**: if the Operator cannot fill all or part of the order, they call `refund()` for the remaining amount, then execute the refund transfer so the leftover USDC is returned to the User's refund address.
4. **Partial fill + refund remainder**: the intended life-cycle allows partial settlement first, followed by refund of whatever part of the order could not be completed.

It should be noted that the protocol has a centralized trust model: when an order is created, the users' funds are moved into a temporary wallet controlled by the protocol. If control over that wallet is lost or abused, users funds may be put at risk.

Code Assessment. The Centiiv Protocol developers provided the source code of the Centiiv Protocol contracts for the code review. The source code appears to be mostly original code written by the Centiiv Protocol developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Centiiv Protocol developers were very responsive and answered queries promptly. The source code contained a test suite, which

the Veridise security analysts noted covered several intended and happy paths, but did not thoroughly test unexpected interactions, invalid inputs, or edge cases.

Summary of Issues Detected. The security assessment uncovered 11 issues, with 3 classified as critical or high severity by the Veridise analysts. Among these, V-CENT-VUL-001 highlights that a user-controlled exchange rate can be submitted without binding it to operator-approved quote data, which may allow users to extract excess cash if downstream settlement relies on the on-chain rate. In addition, V-CENT-VUL-002 shows that invalid `settle_percent` values can drive order accounting into an invalid state, potentially producing negative remaining amounts and corrupting settlement state, while V-CENT-VUL-003 shows that repeated `settle()` calls can overwrite pending settlement records and desynchronize the contract's internal accounting from the actual funds held in the temporary wallet.

The Veridise analysts also identified 2 medium-severity issues, including V-CENT-VUL-004, where partial settlement followed by refund can overcharge users because fees are not tracked against a single remaining fee budget, and V-CENT-VUL-005, where accepting negative refund fees can inflate refund amounts or leave refunds permanently stuck.

In addition, several lower-severity issues were identified, including V-CENT-VUL-007, which notes that freshly created orders may become temporarily locked while the protocol is paused, V-CENT-VUL-008, where fee rates may change during an order's lifetime and lead to unpredictable charges, and V-CENT-VUL-009, where zero-amount settlements can still reduce the recorded remaining basis points and weaken accounting integrity.

The assessment also identified warning-level findings such as V-CENT-VUL-010, which shows that refund and settlement paths are not truly mutually exclusive in practice, and V-CENT-VUL-011, which groups several maintainability concerns that may increase operational risk and make the codebase harder to reason about over time. The Centiiv Protocol developers have been notified and have addressed these findings.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Centiiv Protocol.

Prepare a detailed threat model and perform a follow-up review. The protocol would benefit from a dedicated threat model that clearly defines its actors, trust boundaries, critical assets, off-chain dependencies, and expected workflow. During this review, some important assumptions only became clear after repeated clarification, and at least one meaningful issue was identified only after those broader system details were understood. Once the threat model is documented and agreed upon, the team should consider a second round of security review, ideally with fresh reviewers.

Improve test coverage. The current test suite does not appear to cover the full order life-cycle in enough depth. The protocol would benefit from tests for all intended state transitions and use-cases, including partial settlement, full settlement, partial settlement followed by refund, pause-related flows, and all relevant negative cases such as invalid percentages, duplicate pending actions, and invalid fee values.

Strengthen operational monitoring and recovery procedures. Since the workflow depends on operator-controlled temporary wallets and off-chain actions, the protocol should have clear operational safeguards in addition to smart contract checks. This includes regular reconciliation between on-chain order state and actual wallet balances, alerts for stuck orders or unmatched pending

actions, and a documented recovery procedure for cases where an order cannot be completed normally.

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
Centiiv Protocol	`d758437`	Rust	Stellar

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Apr. 13–Apr. 15, 2026	Manual & Tools	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	2	2	2
Medium-Severity Issues	2	2	2
Low-Severity Issues	4	4	4
Warning-Severity Issues	2	2	2
Informational-Severity Issues	0	0	0
TOTAL	11	11	11

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Data Validation	3
Usability Issue	2
Maintainability	2

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 code-base 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.

3.2 Identified Security Risks

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

- Can protocol flow be broken by executing calls out of the expected order or with unexpected parameter values?
- Can order state transitions lead to inconsistent accounting between the recorded order state, pending actions, and the funds actually held in temporary wallets?
- Can partial settlement, repeated settlement, or refund-after-settlement drive the protocol into states that are not handled correctly by the intended state machine?
- Can extreme or invalid input values for amounts, percentages, or fees break assumptions in the settlement or refund logic?
- Can fee calculation or fee updates during an active order lead to unfair, inconsistent, or unpredictable fee outcomes for users?
- Can pause or emergency controls leave orders in a state where user funds cannot be settled or refunded through the normal protocol flow?
- Can pending actions become stale, overwritten, or executable in an unexpected way, causing value to be lost, delayed, or misaccounted?
- Can trusted offchain roles such as the relayer or temporary wallet controller abuse their authority to delay execution, misroute funds, or lock user funds?
- Can replay, rebroadcast, or timing-related behaviors cause orders to be created or progressed in a way that the protocol does not intend?
- Can auxiliary components or metadata fields create a false sense of security if they are present in the design but not fully enforced by the live order flow?

3.3 Scope

The scope of this security assessment is limited to a specific set of source files from the repository, as agreed upon with the Centiiv Protocol developers:

- `contracts/liquidity_manager_contract/src/error.rs`
- `contracts/liquidity_manager_contract/src/lib.rs`
- `contracts/liquidity_manager_contract/src/liquidity_manager.rs`
- `contracts/liquidity_manager_contract/src/storage.rs`
- `contracts/liquidity_provider_contract/src/error.rs`
- `contracts/liquidity_provider_contract/src/lib.rs`
- `contracts/liquidity_provider_contract/src/liquidity_provider.rs`
- `contracts/liquidity_provider_contract/src/liquidity_provider_trait.rs`
- `contracts/liquidity_provider_contract/src/storage_types.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 the 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 Centiiv Protocol.

- The Operator controls both the relayer and each temporary wallet address (TWA), and is able to authorize actions on behalf of both.
- A fresh TWA is generated for each order and is intended to hold only the USDC associated with that order until the order is either settled or refunded.
- The refund address supplied during order creation is controlled by the user and correctly represents the intended recipient of any refund.
- The Operator is trusted for **liveness**: once funds reach the TWA, the Operator is expected to eventually either settle the delivered portion of the order or refund the unfilled remainder. The contracts do not enforce this behavior.
- The Operator is expected not to move funds out of the TWA through off-protocol or manual transfers. If TWA balances are modified outside the intended settlement and refund flows, onchain accounting may no longer reflect the true location of user funds.
- The offchain cash leg is not verified onchain. Calls to `settle()` and the value of `settle_percent` are assumed to honestly reflect the portion of the user's cash-out that has actually been completed.
- The `liquidity_provider` address supplied during settlement is assumed to be the correct payout destination determined by the Operator, since the contract does not meaningfully authenticate that address against prior order data or a trusted registry.
- Partial settlement followed by refund of the remaining amount is an intended business flow and was treated as such during the review.
- The fields `rate` and `message_hash` are currently treated as metadata only and were not assumed to provide any security, authorization, or validation guarantees.
- Privileged roles, especially the admin / contract owner, are assumed to use pause, fee-update, and upgrade powers according to protocol policy unless a specific finding explicitly discusses misuse or compromise of those roles.

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* 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 role holders carry out their responsibilities as intended. Protocol exploits relying on the roles below acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, emergency roles:
  - **Operator-controlled relayer** can initiate `settle()` and `refund()`, decide when an order progresses, and specify settlement progress through `settle_percent`.
  - **Operator-controlled temporary wallet holder** is trusted to complete the transfer leg of each order instead of leaving funds idle or moving them outside the intended workflow.
  - **Admin / contract owner** can use emergency controls such as `pause()` and `unpause()` to respond to operational incidents.
- Highly-privileged, non-emergency roles:
  - **Admin / contract owner** can initialize the contracts, update protocol addresses such as the treasury and relayer, update fee tiers, and upgrade contract code.
  - **Operator** is trusted to provision fresh TWAs for new orders and to manage the offchain cash side of the workflow in a way that matches the onchain order lifecycle.
- Limited-authority, non-emergency roles:
  - **User / sender** can create and fund orders, choose the refund address, and supply order metadata such as `order_id`, `rate`, and `message_hash`.

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 code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-CENT-VUL-001	User-controlled rate can be used to extract . . .	Critical	Fixed
V-CENT-VUL-002	Invalid settle_percent can break order . . .	High	Fixed
V-CENT-VUL-003	Repeated settle calls overwrite pending . . .	High	Fixed
V-CENT-VUL-004	Partial settlement followed by refund can . . .	Medium	Fixed
V-CENT-VUL-005	Negative refund fee can lock or inflate . . .	Medium	Fixed
V-CENT-VUL-006	Long-lived storage entries do not have in- . . .	Low	Fixed
V-CENT-VUL-007	Paused fresh orders become temporarily . . .	Low	Fixed
V-CENT-VUL-008	Fee rate can change in the middle of order . . .	Low	Fixed
V-CENT-VUL-009	Zero-amount settlement can still reduce . . .	Low	Fixed
V-CENT-VUL-010	Refund and settlement are not mutually . . .	Warning	Fixed
V-CENT-VUL-011	Maintainability concerns	Warning	Fixed

5.1 Detailed Description of Issues

5.1.1 V-CENT-VUL-001: User-controlled rate can be used to extract excess cash

Severity	Critical	**Commit**	d758437
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs`		
Confirmed Fix At	protocol-contracts/pull/39, 13f3887		

Description

The rate field is currently supplied by the user during `create_order()` and is stored and emitted by the contract without any proof that it matches the quote originally issued by Centiiv's backend.

This creates a quote-integrity problem. A possible flow is:

1. Centiiv's backend gives the user-facing app a fresh TWA and an expected exchange rate.
2. Instead of submitting that exact quote, the user or a malicious client submits the same TWA with a higher rate.
3. The contract accepts the order and emits the modified rate in the event.
4. If the cashier or backend later relies on the on-chain order or event as the source of truth, the user can receive more cash than they were supposed to.

In other words, the protocol treats rate as trusted business input even though it is user-controlled.

Impact

If Centiiv's operational flow relies on the stored or emitted rate when settling the cash side, a user can submit an overpriced order and cash out their USDC on much better terms than intended, with the loss falling on Centiiv.

Recommendation

Do not treat the user-supplied rate as an authoritative quote.

Instead, bind the order to operator-approved quote data. For example:

- require an operator signature over the relevant order fields, including rate, amount, TWA, refund address, and expiry
- or use a backend-issued quote ID and verify it off-chain before any cash payout or settlement decision
- or remove rate from security-critical logic entirely and ensure cash settlement is based only on Centiiv's internal quote records

At minimum, the cashier and backend should never rely only on the on-chain rate or `OrderCreated` event when deciding how much cash to deliver.

Developers Response

The developers provided a fix in commit `13f3887` that requires Relayer approval and signature for the exact `create_order()` invocation, preventing users from submitting or altering order fields, such as increasing the rate, without proper authorization.

5.1.2 V-CENT-VUL-002: Invalid settle_percent can break order accounting

Severity	High	**Commit**	d758437
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs:183`		
Confirmed Fix At	protocol-contracts/pull/30, 957eaa2		

Description

It was identified that after a partial settlement, the `settle()` function does not verify that `settle_percent <= order.current_bps`. It only checks that `settle_percent` is within `(0, 100_000]`. As a result, a later settlement can exceed the remaining unsettled portion, causing both `order.current_bps` and `order.amount` to become negative.

Impact

The order's accounting can become invalid, making the remaining state inconsistent with the actual escrowed funds and potentially preventing correct settlement or refund processing.

Recommendation

It is recommended that before applying a settlement, a check for `settle_percent <= order.current_bps` is implemented. Use checked arithmetic for the subsequent amount and BPS updates to prevent invalid state transitions.

Developers Response

The developers provided a fix in commit `957eaa2` that rejects settlement requests when `settle_percent` exceeds the order's remaining `current_bps`.

5.1.3 V-CENT-VUL-003: Repeated settle calls overwrite pending settlement and break order accounting

Severity	High	**Commit**	d758437
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs:183-188`		
Confirmed Fix At	protocol-contracts/pull/29, 750281f		

Description

`settle()` can be called more than once before `execute_settlement_transfer()` is called. The problem is that each new `settle()` call overwrites `PendingSettlement(order_id)` with a new value, but the contract still keeps updating `order.amount` and `order.current_bps` as if the earlier pending settlement still exists.

Because of that, the order can keep moving forward through multiple settlement steps, and can even end up with `is_fulfilled = true`, while only the very last pending settlement remains stored. All earlier pending settlement data is lost.

In the end, the amount still sitting in the temporary wallet address (TWA) no longer matches the last stored settlement slice. The contract state says the order has already progressed much further than what can actually be executed through the remaining pending record.

Impact

This breaks the order accounting badly. The contract can think most or all of the order has already been settled, while the only executable settlement is the final slice that overwrote the previous ones.

As a result, funds can become stuck in the TWA without a matching onchain record that explains how they should be paid out. That can leave the order in a state where it looks completed, but the actual payout history is incomplete and the stored accounting no longer makes sense.

Recommendation

Do not allow `settle()` to create a new pending settlement if one already exists for the same order. The correct approach would be to require the current pending settlement to be executed or cleared first before another `settle()` call is allowed. That keeps `order.amount`, `order.current_bps`, and the stored pending settlement record in sync throughout the order lifecycle.

Developers Response

The developers provided a fix in commit `750281f` that prevents `settle()` from creating a new pending settlement when one already exists for the same order.

5.1.4 V-CENT-VUL-004: Partial settlement followed by refund can overcharge users

Severity	Medium	**Commit**	d758437
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs`		
Confirmed Fix At	protocol-contracts/pull/32, e01a438		

Description

`create_order()` stores a single `order.protocol_fee`, but the contract does not enforce it as the total fee for the order. `settle()` recalculates fees each time on settlement slices, while `refund()` can later charge another fee capped only by the original `order.protocol_fee`, without accounting for fees already taken during prior settlements.

Impact

An order can be charged more total fee than its original implied fee, once it goes through a partial-settlement-then-refund flow. This creates inconsistent accounting and can overcharge users depending on execution path.

Recommendation

Track a per-order remaining fee budget explicitly. Decrease it on each settlement fee, and only allow `refund()` to charge from the unused remainder.

Developers Response

The developers provided a fix in commit `e01a438` where they implemented explicit per-order fee budget tracking, decrementing it on each settlement and restricting `refund()` to charge only from the remaining unused fee amount.

5.1.5 V-CENT-VUL-005: Negative refund fee can lock or inflate refund amount

Severity	Medium	**Commit**	d758437
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs`		
Confirmed Fix At	protocol-contracts/pull/31, da1bb92		

Description

It was identified that `refund()` accepts `fee` as a signed integer and only checks that it does not exceed `order.protocol_fee`. It does not enforce `fee >= 0`. Because `refund_amount` is computed as `order.amount - fee`, a negative fee increases the refund amount instead of reducing it.

Impact

A negative fee can stage a refund larger than the order's remaining amount. If the temporary wallet is strictly isolated per order, this likely causes a stuck refund state. If it can hold unrelated balance, it may also result in over-refund from excess funds.

Recommendation

Validate that `fee` is non-negative before creating `PendingRefund`.

Developers Response

The developers provided a fix in commit `da1bb92` where a check (and a related error) was added to ensure fee is not less than 0.

5.1.6 V-CENT-VUL-006: Long-lived storage entries do not have in-contract TTL extension helpers

Severity	Low	**Commit**	d758437
Type	Usability Issue	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_manager.rs`		
Confirmed Fix At	protocol-contracts/pull/37, c48521e, 632be21		

Description

The contract does not provide helper functions to extend TTL for long-lived control/config storage entries. This is mainly relevant for entries such as `Admin`, `Treasury`, `Relayer`, and `Paused`.

Impact

This is primarily an operational and maintainability concern. Since these entries are low-churn but important for normal protocol operation, relying entirely on external TTL management increases the chance of accidental expiry or overlooked maintenance.

Recommendation

Consider adding lightweight TTL-extension helpers for long-lived state, or make TTL extensions for these entries in the code.

Developers Response

The developers provided a fix for the mentioned issue in `632be21`, along with TTL handling in the LP contract. It now refreshes `SettingsContract`, `Usdc`, and `Admin`, which are the main long-lived LP-side configuration entries, in `c48521e`.

5.1.7 V-CENT-VUL-007: Paused fresh orders become temporarily locked

Severity	Low	**Commit**	d758437
Type	Usability Issue	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs`		
Confirmed Fix At	protocol-contracts/pull/35, 749470b		

Description

If the protocol is paused after a new order is created, the funds for that order are already moved in the temporary wallet address. While the protocol is paused, the contract does not allow either `settle()` or `refund()` to be called. This means a newly created order cannot be neither completed nor canceled through the normal contract flow. If there is no existing `PendingSettlement` or `PendingRefund`, the funds simply stay in the temporary wallet until the protocol is unpaused.

Impact

User funds can get stuck during a protocol pause. This makes the pause mechanism unfair to users. In practice, users may have to wait long time for the protocol to be unpaused or depend on manual operator action outside the contract to get their funds back.

Recommendation

Add an `emergency_refund()` function for paused mode. It should:

- only work while the protocol is paused
- only work if is_fulfilled == false and is_refunded == false
- only work if there is no live PendingSettlement
- send funds only to the saved refund_address
- not allow the operator to choose another receiver
- ideally not allow the operator to charge an extra refund fee

Developers Response

The developers provided a fix in commit `749470b` where they added a paused-only `emergency_refund()` path to allow zero-fee refunds for unfulfilled orders while preventing the flow when a live `PendingSettlement` already exists.

5.1.8 V-CENT-VUL-008: Fee rate can change in the middle of order settlement

Severity	Low	**Commit**	d758437
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs:519-520`		
Confirmed Fix At	protocol-contracts/pull/33, 75a30f5		

Description

The fee tier for an order is not really fixed when the order is created. `settle()` recalculates the fee for each settlement slice using the current tier fee percent, and that value can be changed in the meantime through `update_tier_fee()`.

This means the same order can be charged using one fee rate during an earlier settlement call and a different fee rate during a later one. Because of that, the total fee paid by the user may end up being higher than what they effectively signed up for when the order was created.

Impact

Users do not get a stable fee model for the full lifetime of their order. If the fee percent is changed while the order is still being processed, the final amount taken in fees can become unpredictable and unfair.

This also makes the accounting harder to reason about, because an order no longer has one clear fee rate tied to it from start to finish.

Recommendation

Save the tier fee percent at the time of `create_order()` and keep using that same saved value for all later settlement steps of that order. That way, each order keeps the fee terms it started with, even if the global tier fee is updated later.

Developers Response

The developers provided a fix in commit `75a30f5` where they snapshotted `fee_bps` into the `Order` and updated `settle()` to use that stored value instead of reading the live tier configuration at settlement time.

5.1.9 V-CENT-VUL-009: Zero-amount settlement can still reduce remaining BPS

Severity	Low	**Commit**	d758437
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs:235-236`		
Confirmed Fix At	protocol-contracts/pull/34, 96148c8		

Description

For the `settle()` call, if `settle_percent` is very small, the contract can compute a settlement slice where `liquidity_provider_amount == 0`, but it still decreases `order.current_bps`.

For example, when `settle_percent = 1` and `current_order_bps > order.amount`, this line:

```
liquidity_provider_amount = (order.amount * settle_percent) / current_order_bps
```

rounds down to 0, while `order.current_bps -= settle_percent` still reduces the remaining BPS. So the contract records settlement progress even though no value was actually moved out of the order.

Impact

This breaks the link between current_bps and the real remaining amount. The order can look partially settled even though the full amount is still sitting in the temporary wallet. With the current USDC-only setup, this is less practical because `current_order_bps` is capped at `100_000`, which is only `0.01` USDC in 7-decimal units. But the behavior is still wrong today, and it becomes much more dangerous if the protocol later supports tokens with fewer decimals or tokens where each base unit is worth more.

Recommendation

Do not allow a settlement step that computes a zero payout.

At minimum, reject the call if `liquidity_provider_amount == 0`. It would also make sense to enforce `settle_percent <= order.current_bps` so the settlement math stays inside its intended range.

Developers Response

The developers provided a fix in commit `96148c8` where they implemented the recommendation and added a check to reject cases where `liquidity_provuider_amount == 0`.

5.1.10 V-CENT-VUL-010: Refund and settlement are not mutually exclusive

Severity	Warning	**Commit**	d758437
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/[...]/liquidity_provider.rs:367-368`		
Confirmed Fix At	protocol-contracts/pull/38, 6fbce1a		

Description

The contract allows the following sequence:

1. `create_order()`
2. `partial settle()`
3. `refund()`
4. `execute_settlement_transfer()`
5. `execute_refund_transfer()`

After a partial `settle()`, the contract stores a `PendingSettlement`. Later, `refund()` stores a `PendingRefund`, marks the order as refunded, and zeros out `current_bps` and amount. However, it does not clear the existing `PendingSettlement`, so the old settlement can still be executed afterward.

This behavior does not match the documented state model. In `storage_types.rs`, the comments say settlement and refund states are mutually exclusive and one-way. That is not how the implementation behaves: one order can go through partial settlement and then refund the remaining amount, with both transfer paths still executing.

Impact

The main issue is that the order state becomes misleading. An order marked as `is_refunded = true` may still have a settlement leg that can execute later. This makes it harder for developers, integrators, and offchain systems to understand what the order state really means. That kind of mismatch can easily lead to wrong assumptions in monitoring, indexing, reconciliation, and future code changes. Even if the current fund flow is economically valid, the contract does not express that flow clearly.

Recommendation

If this is intended behavior, then the comments and state naming are wrong and should be fixed.

If this is not intended, then the logic should be changed to enforce the intended lifecycle.

Developers Response

The developers provided a fix in commit `6fbce1a` where they clarified the intended partial-settle-then-refund behavior in the documentation and updated `refund()` to reject when a live `PendingSettlement` exists.

5.1.11 V-CENT-VUL-011: Maintainability concerns

Severity	Warning	**Commit**	d758437
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/` ▶ `liquidity_manager_contract/src/liquidity_manager.rs` ▶ `liquidity_provider_contract/src/` • `liquidity_provider.rs` • `storage_types.rs`		
Confirmed Fix At	protocol-contracts/pull/36, 010fad9		

Description

The following maintainability concerns were identified throughout the codebase:

Misleading or incorrect comments:

1. **liquidity_provider_contract/src/storage.rs@L132-133**:
   The "non-custodial" wording is misleading, as the current design assumes that the temporary wallet is controlled by the protocol/backend rather than by the end user.
2. **liquidity_manager_contract/src/liquidity_manager.rs@L140-141:**
   The comment states that existing orders can still be settled or refunded while paused. In practice, `settle()` and `refund()` cannot be called while the protocol is paused, and the tests appear to confirm that behavior. This suggests the comment is inaccurate.
3. **liquidity_provider_contract/src/liquidity_provider.rs@513-514:** The comment around `calculate_protocol_fee()` is misleading, as the implementation does not introduce any additional internal decimal precision.

Unused program constructs

1. **liquidity_manager_contract/src/liquidity_manager.rs@L54-56:** `MaxBps` is stored but never actually used, while the literal `100_000` is hardcoded elsewhere.
2. **liquidity_manager_contract/src/liquidity_manager.rs@L627-656:** `register_lp_node()` does not appear to participate in meaningful protocol logic. The developers indicated that it is intended only for tests; if so, it should be isolated accordingly, for example with `#[cfg(test)]`.
3. **liquidity_provider_contract/src/liquidity_provider.rs@125-136:** `nonces` appear unnecessary, as they are neither consumed by the protocol logic nor needed for replay protection in the current design. The order ID already serves as the relevant uniqueness mechanism.

Miscellaneous

1. **liquidity_manager_contract/src/liquidity_manager.rs@L624-625, L607-608:**
   Literal numeric values are used directly instead of named constants, which reduces readability and increases the risk of inconsistent updates.
2. **liquidity_manager_contract/src/liquidity_manager.rs@L122:** The terminology is inconsistent: when `ProtocolAddressType::Aggregator` is selected, the `Relayer` storage entry is updated. This is confusing and can lead to misunderstanding of role boundaries.
3. **liquidity_manager_contract/src/liquidity_manager.rs@L52:**
   Once an admin address is set, it can no longer be updated in case of emergencies. An alternative would be to use multisig for the admin, as Stellar supports this out of the box.

4. **liquidity_manager_contract/src/liquidity_manager.rs@L43:** Using `__constructor()` for initialization would reduce the risk of initialization frontrunning and provide a clearer deployment pattern.

Impact

The mentioned maintainability concerns may degrade code quality, negatively impact long term maintainability or mislead users.

Recommendation

It is recommended to address the mentioned maintainability concerns.

Developers Response

The developers fixed most of the findings.