certora

# Security Assessment Report

# Blend

January 2024

***Prepared for***
Script3 Ltd

certora

# Table Of Content

certora

# Project Summary

The purpose of this research was to audit all components of the Blend protocol, with the goal of validating the architecture and design, and strengthening any assumptions of soundness, security and robustness of the DeFi application.

## Stellar and Soroban

Stellar is a blockchain platform, on which Soroban provides a smart-contract runtime.
Soroban is powered by a WASM VM, and provides a toolchain for developing and deploying contracts written in Rust.

At the time of this document's writing, Soroban is still in very early stages, and as such there exists very little material, both in terms of existing contracts and in security research.

Soroban provides a fairly simple authentication interface, allowing contract developers to validate that specific addresses have signed (fully or partially) for the current invocation.

Other environmental interfaces exist in Soroban, for on-ledger persistent and temporary storage, as well as various arithmetic and cryptographic primitives.

All in all - Soroban contracts are idiomatic from the perspective of a Rust developer, and require significantly less scrutiny with regard to unusual code and in-code configurations, compared to Solana for example.

## Blend

Blend is a lending protocol implemented on Soroban, which implements the logic for several components of a liquidity pool - namely an emitter, responsible for the generation of rewards in the form of BLND tokens, liquidity pool instances, and a backstop which provides insurance for the liquidity pools.

Its design is oriented towards autonomous operation and includes very basic governance features, aimed to promote sustainability and incentivize development of the protocol.

The protocol consists of several components, represented as sub-contracts within the code base.

The emitter is responsible for minting BLND, which is the token deposited in the backstop module for use as insurance, and as a reward for participating actors in both the backstop module and the lending pools.

The backstop serves as insurance for lending pools, maintaining a separate balance for each lending pool, which is used as first-loss capital for any bad debt associated with its respective lending pool.
Backstop depositors acquire shares within the pool by depositing a USDC/BLND LP token, which they can withdraw subject to a 21-day locking period.
In return, they earn a portion of any interest accumulated within the pools, as well as BLND rewards in some cases.

The liquidity pool factory is a contract that deploys liquidity pools while affixing certain parameters that are intended to be immutable: the bytecode of the contract for the pool, the address of the backstop module, and the BLND and USDC token addresses.

The liquidity pool is a contract that implements the underlying lending/borrowing logic, as well as managing user states and balances, liquidating users via an auction mechanism, as well as more complex interactions with the backstop such as transferring and selling off bad debt, and exchanging accumulated backstop interest for funds.

The liquidity pool contract manages the lending/borrowing and supply logic by keeping a balance in bToken and dTokens, representing incremental indices through which users earn and pay interest, respectively.
This is similar to the way Aave and other lending protocols track user positions.
A key aspect in which Blend is different from Aave is the way interest rates are calculated - Blend calculates the effective interest rate dynamically based on utilization rates, similar to Aave, but novel in the sense that the interest rate scales differently based on segregation of the utilization state into 3 distinct regions, of different orders and with different coefficients.

## Code Base

Usually, our audits are performed on a frozen code base, pinned to a certain commit (commit hash: 5a64f8b9a8472b232f9fefe861b2638993b894c8)
However, the Blend code base is still under continuous development, and we performed our research on the live repository.

## Timeline

The research was performed during 6 weeks between October and December 2023 (Dec 7).

# Audit Methodology

The research was conducted in two main phases:

1. First reading - this phase, which lasted about 3 weeks, consisted of a top-level reading of the entire code base, and was intended to provide a good understanding of the various components and subcontracts and their interactions, as well as pointing out vulnerabilities and anti-patterns on the surface.
2. Deep dive - this phase, which lasted about 3 weeks, included a more in-depth review of components within the system that were deemed to be high-risk/high-value attack surfaces.

## Results

**This audit resulted in several high-severity findings, as well as multiple recommendations in both architecture and implementation.**

All raised issues were addressed by the Blend dev team.

The report is divided into two main sections. The findings section contains all major issues raised, while the recommendations section contains other actionable observations and suggestions which in our view will improve the protocol and quality of code.
This includes general and Rust-idiomatic best practice recommendations.

certora

# Detailed Findings

| Identifier | Description | Severity | Location |
|---|---|---|---|
| BL-001 | “Dead-end” flows (resource limits) | CRITICAL | pool - health check, others |
| BL-002 | BLND reward loss during interval between emission cycles | MEDIUM | backstop - update_emissions_cycle |
| BL-003 | Incomplete validation of contract types in backstop deposit flow | HIGH | backstop - execute_deposit |
| BL-004 | Unearned reward emission in intersection of reward zone growth and emissions cycle update | INFO | backstop - add_to_reward_zone, update_emissions_cycle |
| BL-005 | Lack of validation that filler state is different from the user address in fill_user_liq_auction | LOW | pool - fill_user_liq_auction |
| BL-006 | Incorrect bounds check of reactivity constant in require_valid_reserve_metadata | INFO | pool - require_valid_reserve_metadata |

## BL-001 - "Dead-end" flows (resource limits)

This issue was raised by the Blend developer team at an early stage of our audit, and independently from it.

They discovered that under specific flows and given certain user states (balance, positions) - the Soroban resource limit is invariably hit, causing some code and underlying logic to be impossible to execute.

In the provided scenario - the most resource-intensive code pertained to the health check, which reads the asset reserve information from ledger storage, resulting in significant IO utilization for each supported asset in the pool.

The impact, as illustrated by the developers - could be lack of ability to liquidate users who hold a large number of assets.

Since the number of assets scales IO utilization linearly, we suggested setting a hard limit to the number of supported assets per pool.

From a more strategic perspective - we suggest adding the following steps to the development and testing cycle:

1. Setting a high watermark at 80-90% of the Soroban resource limit, which will serve as a safety buffer
2. Profiling all unit tests to ensure they remain under the high watermark
3. For the health check flow and other flows deemed problematic - determine the scaling factors - for instance, the amount of asset reserve balances held by one user - and limit them in code.

**Blend's response:** Acknowledged and fixed

## BL-002 - BLND reward loss during intervals between emission cycles

The weekly emissions cycle is triggered manually upon a contract-call to the backstop module.
The emitter module mints 1 BLND per second, which it deposits to the backstop independently of the backstop emissions cycle update.

During the emissions cycle update - the last update time is tracked by the ledger timestamp - if the current timestamp is less than a week after the currently saved timestamp, the update is aborted.

Otherwise, the emissions per second and subsequently the BLND reward balance is determined for each pool in the reward zone, according to its share in the backstop.

An issue arises when the emissions cycle update is invoked later than exactly 1 week after the previous one - since in any "gap" period, BLND that is minted and distributed by the emitter will be left inaccessible by the backstop - since it can only distribute a week's quanta of BLND in discrete, nonoverlapping periods, which have arbitrary start and end periods.

For example - consider the following scenario:

1. At point T: The emissions cycle is updated, setting T to be the last update time
2. At point T + 1 week: the emissions cycle is NOT updated for some reason
3. At point T + 2 weeks, the emissions cycle is updated, causing 1 weeks worth of BLND to be distributed from the backstop to the pools

In this scenario - during the two weeks between T and T+2 - only 1 weeks worth of BLND was distributed from the backstop to the pools, but 2 weeks worth of BLND was distributed from the emitter to the backstop - resulting in 1 week of BLND becoming "stuck" in the backstop.

Our recommendation is to consolidate the calculation so it is tracked with an index that is directly coupled to the time passed - that way there will be no discrepancies between the BLND per second calculated in the emitter and the backstop.

This issue was raised in real time during our review, and was already planned for mitigation at the time of this document's writing.

**Blend's response:** Acknowledged and fixed

## BL-003 - Incomplete validation of contract types in backstop deposit flow

When depositing to the backstop, the address of the pool for which the funds will serve as insurance is provided as part of the contract call.
However, there is no validation that the address belongs to a pool belonging to the same backstop instance, or to any pool at all.

As a result - it is possible to deposit to the backstop, and potentially earn interest and rewards without ever risking any funds.

Practically, this does not lead to an attack due to parallel constraints in other contract flows which need to run for interest and rewards to be paid out.

However, we recommend adding an explicit check that the pool address to be deposited to is indeed a pool instantiated by the correct factory.

This issue was raised in real time during our review, and was already planned for mitigation at the time of this document's writing.

**Blend's response:** Acknowledged and fixed

## BL-004 - Unearned reward emission in the intersection of reward zone growth and emissions cycle update

This is a very minor, anecdotal issue related to the way the reward and emissions cycles.
Usually, when a pool is entered into the reward zone, it is done no sooner than 5 days prior to the emissions cycle weekly update.

This ensures that backstop depositors cannot manipulate their balances in order to enter or remain in the reward zone.
However, this time lock does not apply to new slots in the reward zone, which open up every 97 days.

As such, an actor can time the intersection of these two cycles to enter the reward zone without necessarily holding eligible amounts of funds.

We recommend that the time lock be applied to entry into open slots in the reward zone, even if it means they remain empty for that time lock period.

**Blend's response:** Acknowledged and fixed

## BL-005 - Lack of validation that filler state is different from the user address in fill_user_liq_auction

The fill_user_liq_auction function is responsible for transferring outstanding liabilities and collateral from the liquidated user to the liquidator.

We recommend adding a check here that the liquidator address is not the same as the liquidated address.

While it is currently impossible to create a situation where this occurs - relatively minor changes to the code in the future could expose this issue to exploitation.

**Blend's response:** Acknowledged and won't fix, see issue

## BL-006 - Incorrect bounds check of reactivity constant in require_valid_reserve_metadata

According to the whitepaper, the reactivity constant should be bounded from 0.001 to 0.00001. However, the code in require_valid_reserve_metadata only checks that it is above 0.0005.

If this is intentional, a comment should be added or the whitepaper could be clarified further.

**Blend's response:** Acknowledged and fixed

# Recommendations

The following do not constitute issues that require immediate fixes in our view, but are still recommended as actions to improve the quality, robustness and soundness of the codebase, and will assist in maintaining its security over time.

## BLRC-001 - Inconsistent use of scalar constants

Throughout the code base, scalar constants are used inconsistently - at times constants from the constants.rs files are used, and at times the numerical values are used.

Numerical values are error-prone, and an omitted digit can have a disastrous effect - as such we recommend replacing all numerical scalar values wherever they are used in favor of the already in-place constants.
**Blend's response:** Acknowledged and fixed

## BLRC-002 - Potential pitfalls in reserve loading and caching mechanism

This is a design note - at the moment, seemingly as an optimization to avoid performing expensive IO operations - the reserve data is read from storage once, and then cached for later reads.

To perform a writeback, i.e. write the "dirty" state back to ledger storage, one must invoke `cache_reserve` with the write flag set.

In our view, this is an error-prone design, and future changes to the code may omit the writeback operation, or add "escape paths" in which the transaction may exit successfully before the writeback occurs.
In other cases - a developer may add the writeback when is it not required.

As such we recommend implementing a new design for the reserve caching system.

One option that we suggest is adding a drop guard pattern in which the reserve structure is validated to have been written back to storage unless it has been explicitly "defused".
**Blend's response:** Acknowledged and fixed

## BLRC-003 - Inconsistent use of enum discriminant constants

Throughout the code there are multiple locations in which the numerical values of the enum discriminant are used instead of the semantic matches.

This is an anti-pattern and may cause serious issues to go unnoticed in case the enums are updated in future versions.

On the same note - the `Request` struct and its consuming function `build_actions_from_request` rely on a pure numerical value to determine the request type.

We recommend refactoring all comparisons and matches utilizing fixed numerical values and using the enum semantic values instead.
**Blend's response:** Acknowledged and fixed

## BLRC-004 - Non-idiomatic initialization check in various initialization flows

During the initialization flows for the pool contract, the emitter contract, and the backstop contract, the current deployment is tested to ensure it is not being re-initialized, by checking certain parameters that would have been keyed into storage in the initialization flow.

For example, the pool contract checks if the admin has been set,

While this in itself is not an issue, it constitutes secondary logic that does not directly represent the initialization state.
Instead, we recommend adding a dedicated storage field keyed specifically to the initialization state ("IsInitialized")
**Blend's response:** Acknowledged and fixed

## BLRC-005 - Split dequeue shares function into discrete flows - expired and non expired shares

The `dequeue_shares_for_withdrawal` function is currently responsible for two distinct operations - withdrawal of shares that have passed time-lock, and cancellation of requests to withdraw shares.

We view this design as slightly confusing and potentially prone to misuse.

We recommend separating the two flows - either by splitting them into two functions or creating clearer wrappers.
**Blend's response:** Acknowledged and won't fix, see issue

## BLRC-006 - Add explicit positive `share - q4w` constraint in `update_emission_data_with_config`

As a sanity check, we recommend ensuring the result of the share - q4w calculation in `update_emission_data_with_config` is positive.
**Blend's response:** Acknowledged and fixed

## BLRC-007 - Add explicit `emis_data.index - user_data.index` constraint in `update_user_emissions`

Similarly - we recommend another such sanity check in `update_user_emissions`
**Blend's response:** Acknowledged and fixed

## BLRC-008 - Extract claiming flow from `update_emissions` to a discrete function

Similar to BLRC-005 - the update_emissions function currently consolidates two flows - one for updating emissions in place, and another for performing the update followed by a "claim".

In this design, the claim flag is drilled down multiple functions, and the return value has two different meanings depending on its setting.

This is not a Rust idiomatic design, and we find it to be somewhat confusing and potentially problematic in the future.

We recommend extracting the claim flow to a different function, and invoking it where necessary.
**Blend's response:** Acknowledged and fixed

## BLRC-009 - Add sanity check `ledger_timestamp - token_emission_data.last_time` is positive, in `update_emission_data`

Similar to BLRC-006 and 007 - we recommend adding an explicit sanity check that the time interval calculation in `update_emission_data` always results in a positive value.
**Blend's response:** Acknowledged and fixed

## BLRC-010 - Add signer validation for incoming new admin in `set_admin`

As a measure of extreme caution - when setting a new admin for a pool we recommend requiring that the new admin also signs for the request - to ensure a mistaken address does not cause the admin value to become stuck and potentially brick the pool.
**Blend's response:** Acknowledged and fixed

## BLRC-011 - Lack of payer/payee sanity checks in `execute_deposit`

The transfer performed during this function does not validate that the `from` and `to` addresses are not the same - or in this case that the from address is not the backstop address.
This has no practical impact at this point, but we recommend adding this check for posterity.

**Blend's response:** Acknowledged and fixed

## BLRC-012 - Lack of payer/payee sanity checks in `execute_donate`

The transfer performed during this function does not validate that the `from` and `to` addresses are not the same - or in this case that the from address is not the backstop address.
This has no practical impact at this point, but we recommend adding this check for posterity.

**Blend's response:** Acknowledged and fixed

## BLRC-013 - Lack of validation that filler address differs from the backstop or pool addresses in fill_interest_auction

The `fill_interest_auction` function allows users to "buy off" a pool's backstop credit by sending USDC to the backstop and receiving assets in return.

While this does not currently cause any issues - we recommend adding sanity checks to ensure the filler address is not the backstop or the target pool's address.

**Blend's response:** Acknowledged and fixed

## BLRC-014 - Lack of validation that filler address differs from the backstop or pool addresses in fill_bad_debt_auction

The `fill_bad_debt_auction` function allows users to take on bad debt positions in exchange for backstop tokens.

While this does not currently cause any issues - we recommend adding sanity checks to ensure the filler address is not the backstop or the target pool's address.

**Blend's response:** Acknowledged and fixed

## BLRC-015- Lack of target validation across function boundaries in bad debt burn flow

The function `burn_backstop_bad_debt` in the pool contract accepts a given pool and the mutable backstop state and removes all liabilities from it.

This function assumes that the backstop variable is a handle to the backstop state and does not perform any additional verification.

It is generally recommended not to rely on calling functions to provide security guarantees without providing explicit cautions to developers.

In this case, while at the moment this function is invoked correctly and safely, future changes to the code base will be prone to misuse of this interface.

We recommend either moving the `User::load` call inside the function, or otherwise explicitly verifying that the provided state indeed belongs to the backstop, inside the function.

**Blend's response:** Acknowledged and fixed

## BLRC-016 - Backstop critical state not validated across function boundaries in bad debt burn flow

Similar to the previous finding - since the `burn_backstop_bad_debt` function is always meant to be invoked when the backstop is in a certain state.

As such, we recommend moving the threshold logic into the function, to avoid any future developer error which may cause this function to be misused.

**Blend's response:** Acknowledged and won't fix, see issue

## BLRC-017 - Lack of validation across functional boundaries in auction scaling flow

The `scale_auction` function in the pool contract takes an auction data structure and a percentage to which its bid and lot values must be scaled.

The percentage is verified to be in valid range by the caller to the `scale_auction` function, instead of the function itself.

It is generally recommended not to rely on calling functions to provide security guarantees without providing explicit cautions to developers.

In this case, while at the moment this function is invoked correctly and safely, future changes to the code base will be prone to misuse of this interface.

We recommend moving or duplicating the 0 to 100 percent range check into the `scale_auction` function itself.

**Blend's response:** Acknowledged and fixed

## BLRC-018 - Integer underflow in auction scaling mechanism

During auctions - the amount of collateral earned and amount of liability assumed are respectively earned and decreased, based on time, as a secondary pricing mechanism for all three auction types.

The way time factors into the calculation is by subtracting the current block sequence from the one stored in the contracts ledger storage, for the ongoing auction.

When an auction is created, this stored number is set to be the current block + 1. When the auction is filled, the stored number is subtracted from the current block number.
The issue arises when an auction is created and filled within the same block - since the modifier is calculated as `block_num - (block_num + 1) = -1`
Since the block number is an unsigned integer - this value will underflow to $2^{32} - 1$, which is then used to scale the lot modifier, resulting in an extremely large payout and possible drainage of the contract.

At the moment, this issue is mitigated by compilation flags which ensure that integer boundaries issues are checked against - even in release builds.

We nevertheless recommend correcting this calculation and incorporating explicit `checked_sub/checked_add` arithmetic variants for all calculations in the code.
This ensures that any changes to the build process will not result in the original issue seeping into the deployed contract.

**Blend's response:** Acknowledged and fixed

# Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora 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.

# About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects,
and incident response.