Security Assessment
DRAFT v3 Report
Blend v2

April 2025

Prepared for
Script3 Ltd

certora

Table of contents

certora

Project Summary

Project Scope

Project Name	Repository (link)	Audited Commits	Platform
Blend V2	https://github.com/blend-capital/blend-contracts-v2	589d082 (original commit)	Stellar
		04922e5 (PR#46 - audit fixes)	Stellar

Project Overview

This document describes the manual code review findings of **Blend V2 Contracts**. The following contract list is included in our scope:

- `backstop/src/*`
- `pool-factory/src/*`
- `pool/src/*`

The work was undertaken from **February 03, 2025,** to **March 27, 2025**. During this time, Certora's security researchers performed a manual audit of all the Stellar contracts (commit 589d082) and discovered several bugs in the codebase which are summarized in the subsequent section. In addition, while the audit was taking place, the Blend team discovered a bug related to the handling of rebasing tokens by the gulp function. Fixes for all of the aforementioned issues were merged into a single pull request (commit hash 04922e5) and reviewed by the Certora team.

At the same time, Certora's security engineers, working in tandem with the security researchers, wrote a set of formal rules and properties for Blend's codebase and proved their correctness using the Certora Prover. We share those findings in a separate report.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	2	2	2
Medium	2	2	2
Low	3	3	1
Informational	3	3	2
Total	**10**	10	7

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
H-01	The protocol is vulnerable to sybil attacks due to the lack of restriction on position size	High	Fixed
H-02	Users can create nearly unfillable auctions	High	Fixed
M-01	Missing MAX_POSITIONS check in the flash loan implementation	Medium	Fixed
M-02	Users can use share inflation to bypass partial liquidation protection	Medium	Fixed
L-01	Lack of off-chain infrastructure to call state-changing functions	Low	Acknowledged
L-02	Missing check to ensure reserve.config.util is less than 0.95	Low	Fixed
L-03	Token to shares ratio can be 1:1	Low	Acknowledged
I-01	Missing or Incorrect documentation	Info	Fixed
I-02	Code quality and best practices	Info	Acknowledged
I-03	Spelling Mistakes	Info	Fixed

High Severity Issues

H-01. The protocol is vulnerable to sybil attacks due to the lack of restriction on position size

Severity: **Critical**	Impact: **High**	Likelihood: **Medium**
	Status: Fixed	

Description: The actions in the protocol (deposit, borrow, repay, liquidation, etc.) do not have minimum checks to check how much has been deposited into the system. An attacker, for example, can take advantage of this by splitting a risky position into multiple, smaller positions under different addresses, making each one unprofitable to liquidate in case the position falls under water.

While healthy positions can be liquidatable, liquidators may still lack the incentive to process these liquidations on-chain, especially in high-cost gas environments, where the user may have to pay more gas than the benefit they receive for ridding the system of bad debt.

Example:

1. A user wants to deposit a large amount of funds into the system but is interested in finding a way to reduce the possibility of being liquidated if the position goes under.
2. To limit the chances of liquidation, the user splits this deposit into many smaller increments of $5, across multiple accounts
3. Due to the small size of the position, the costs to liquidate these positions by the sending of gas costs exceeds the rewards the liquidator would receive for them.
4. This results in mismatched economic incentives - that until market conditions make this liquidation more profitable, the bad debt will continue to accumulate.

Remark: Apriori, one might assume that such an attack would be impossible in Stellar since there is a fixed non-trivial cost in XLM to create a new account which serves as built-in mitigation. But this is only true if the attackers intended to create such identities legally.

There is another way - there are many poorly configured user accounts in any Blockchain (e.g., users who have already been rekted by crypto-scammers and essentially abandoned their wallets). We note that while these accounts usually contain low TLV and and therefore not profitable in and of themselves, they can be used as a staging ground to launch sybil attacks against DeFi protocols.

Recommendations: Add a restriction on position size to all user actions in the protocol.

Customer's response: Fixed in PR#46.

Fix Review: Fix confirmed.

H-02. Users can create nearly unfillable auctions

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: • pool/src/pool/user.rs	Status: Fixed	

Description of the issue: The root cause of the problem is that the function remove_liabilities which has to be called while an auction is being filled is using ordinary subtraction instead of saturating subtraction:

```
Unset
pub fn remove_liabilities(&mut self, e: &Env, reserve: &mut Reserve, amount: i128) {
	if amount <= 0
	{
		panic_with_error!(e, PoolError::InvalidDTokenBurnAmount)
	}
	let balance = self.get_liabilities(reserve.config.index);
	self.update_d_emissions(e, reserve, balance);
	let new_balance = balance - amount;
	require_nonnegative(e, &new_balance);
	...
```

Thus, if balance is zero and amount >0, this function will revert.

Description of the potential exploit: Eve wants to make itself virtually immune to liquidations. To do so -

1. Eve borrows a certain amount of reserve X (say 200000 = 2000 * 100 tokens), and continues trading as usual.
2. If Eve's position ever becomes unhealthy (i.e., its health factor falls below 1), Eve pre-emptively creates a liquidation auction on itself, including the liabilities of reserve X.
3. Eve repays the entire liability of reserve X using tokens it kept aside for this purpose.
4. The auction is now unfillable - if Bob tries to fill the auction (using fill_user_liquidation_auction) the call to rm_positions would revert because of an underflow when trying to account for reserve X. Note that even trying to do a partial fill

and paying a certain percentage won't help - the amount we need to pay is zero so even setting percent_filled = 1 and waiting 399 blocks would still trigger the same problem.

5. Eve waits for 399 blocks to pass. At this point the auction now offers 100% of the collateral for only 0.05% of the original liabilities.
6. Eve now takes a quick flash loan (either from this protocol or a different one) and then submits the following requests vector:
7. Borrows 200000 tokens of reserve X again.
8. Fills its own liquidation order, gaining back all the collateral and only taking back a fraction of the liabilities because of the scaling factor.
9. Repays 200000 tokens of reserve X.
10. Since we only had to take back 0.05% of original liabilities our health factor is good again at the end of the build_actions_from_requests check.
11. Eve now repays the flash loan and concludes the transaction, after passing 99.95% of its debt onto the pool.

Remark: A few (slightly technical) subtle points that we would like to emphasize in the description of the exploit above -

- In step 2, Eve has to create the self-liquidation auction on itself via a different key since self-liquidation is currently impossible in Blend V2.
- This is what makes the attack hard, if Eve were to naively follow the steps outlined above, she would have to compete against the market to fill the liquidation after the borrow is confirmed in a different transaction, and risk losing 99% of their collateral. We further note that in this case it is also unlikely for Eve's position to get fully liquidated, and that most liquidations of unhealthy positions are around 10-25%.
- Instead, in order for Eve to carry out such an attack in reasonable safety, Eve has to control at least two keys as well as a portion of the validators to carry out a multi-transaction attack on the SCP.

Recommendation: Replace the subtraction operation with saturating subtraction and adjust the rest of the code accordingly or remove the ability of a user to modify its position while under liquidations.

Customer's Response: Fixed in PR#46 by blocking actions against liquidations (except cancelling the liquidation of course).
Fix Review: Fix confirmed.

Medium Severity Issues

M-01. Missing `MAX_POSITIONS` check in the flash loan implementation

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: • pool/src/contract.rs	Status: Fixed	

Description: The function flash_loan (and execute_submit_with_flash_loan) is missing a check for MAX_POSITIONS. Upon first glance, this seems innocuous enough, until we note that the check in the end of build_actions_from_request:

```
Unset
        ...
        // Verify max positions haven't been exceeded
        pool.require_under_max(e, &from_state.positions, prev_positions_count);
```

which calls pool.require_under_max:

```
Unset
pub fn require_under_max(&self, e: &Env, positions: &Positions, previous_num: u32) {
                let new_num = positions.effective_count();
                if new_num > previous_num && self.config.max_positions < new_num {
                        panic_with_error!(e, PoolError::MaxPositionsExceeded)
                }
        }
```

only reverts upon an *increase* from the previous_num of positions, which are recorded at the beginning of the function:

```
Unset
pub fn build_actions_from_request(
       e: &Env,
       pool: &mut Pool,
       from_state: &mut User,
       requests: Vec<Request>, ) -> Actions {
              let mut actions = Actions::new(e);
              let prev_positions_count = from_state.positions.effective_count();
              ...
```

Recommendation: Add the missing check.

Customer's Response: Fixed in PR#46.

Fix Review: Fix confirmed.

M-02. Users can use share inflation to bypass partial liquidation protection

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
	Status: Fixed	

Description: Blend relies on permissionless and decentralized liquidators to deal with unhealthy positions. In this type of lending and borrowing protocol, partial liquidation is essential to allow large positions to be liquidated in small portions. If it was not for the existence of such an option, large liquidatable positions ("whale liquidation") could remain unresolved for extended periods of time since individual liquidators might lack the required tokens to cover the entire debt.

However, there is a way for malicious actors to bypass the option and create an "all-or-nothing" situation using an attack reminiscent of AAVE's "empty pool" attack vector:

- The attack begins by finding an empty reserve (e.g., a reserve that has just been added to the pool) with underlying token A. For simplicity, assume that `bstop_rate = 0` and that the token A has 18 decimals.
- The attacker (Eve) supplies 1e18 A token to the pool as collateral to and receives 1e18 shares in return. Thus we have b_rate = 1e12 (conversion rate is 1:1 with 12 decimals of accuracy) and b_supply = 1e18.
- Eve then transfers a large amount of tokens (say 1eN * 1e18) directly to the pool and calls the function gulp to update the reserve's bToken rate. Looking at the flow of execute_gulp, we see that `pool_token_balance = (1eN+1)*1e18` while `reserve_token_balance = 1eN * 1e18` so that `token_balance_delta = 1eN * 1e18`. Therefore the new b_rate = [(1e18 + 1eN* 1e18 - 0) * 1e12] / 1e18 = (1eN+1) * 1e12.
- Eve now proceeds to withdraw collateral for (1e18-1)**(1eN+1) tokens, burning 1e18-1 shares.
- At this point, Eve has 1 A-reserve share (i.e., a bToken) which is worth (1eN+1) underlying A-tokens. Thus, we note that Eve hasn't lost (or gained) anything yet.
- If N was taken to be large enough, Eve can now Borrow from other reserves against Eve's single collateral bToken.

- The problem is that if Eve's position ever goes under, it is impossible to partially liquidate it: indeed, any user liquidation which is not a full liquidation is quoted in terms of bTokens and dTokens removed (see L.#154-170) but if b_tokens_removed > 0 it is necessarily a 100% liquidation since all collateral is being removed.

Customer's Response: This issue is resolved by commit 659ed1a which fixed a similar issue with the gulp function related to rebasing tokens.

Fix Review: Fix confirmed.

Low Severity Issues

L-01. Lack of off-chain infrastructure to call state-changing functions

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
	Status: Acknowledged	

Description: The Blend system has many different features that depend on third parties, such as backstop depositors calling functions to change state—for example, if a pool is frozen, removing it from the reward pool, or for users to execute and change pool state.

During periods of time when the gas cost is abnormally high or a particular reserve has much less activity than usual, resulting in fewer depositors watching the pool, the system will not update the pool's state properly. This is dangerous, as pools may be eligible for rewards despite not being active or the state that should change based on q4w values.

Description of the potential exploit:

1. A user with a large stake in a pool decides to withdraw from the system
2. In doing so, the user takes out 70% of an obscure tokens' reserves
3. The pool state should be set to frozen, but all other backstop depositors and other token holders in the pool do not pay close attention to the blockchain, and therefore do not know that they need to update the pool status
4. The pool continues to operate in an "active" status

Customer's Response: Acknowledged.

L-02. Missing check to ensure `reserve.config.util` is less than 0.95

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
	Status: Fixed	

Description:

The pool lacks a check to ensure that the target_util of a pool is less than 0.95. This means that on calculating the accrual of a pool, the calculations will not enter the expected branch to calculate the current interest rate.

The following code is used to set the reserve config:

```
Unset
    let reserve_config = ReserveConfig {
         index,
         decimals: config.decimals,
         c_factor: config.c_factor,
         l_factor: config.l_factor,
         util: config.util,
         max_util: config.max_util,
         r_base: config.r_base,
         r_one: config.r_one,
         r_two: config.r_two,
         r_three: config.r_three,
         reactivity: config.reactivity,
         collateral_cap: config.collateral_cap,
         enabled: config.enabled,
     };
     storage::set_res_config(e, asset, &reserve_config);
```

Not checking the incoming util variable means that if the curr_util of the system is 0.95, and target_util is also 0.95, the code will enter the first if statement, where the curr_ir value is different.

```
Unset
   if cur_util <= target_util {
        let util_scalar = cur_util.fixed_div_ceil(e, &target_util, &SCALAR_7);
        let base_rate =
            util_scalar.fixed_mul_ceil(e, &i128(config.r_one), &SCALAR_7) +
i128(config.r_base);

        cur_ir = base_rate.fixed_mul_ceil(e, &ir_mod, &SCALAR_7);
    } else if cur_util <= 0_9500000 {
        let util_scalar =
            (cur_util - target_util).fixed_div_ceil(e, &(0_9500000 - target_util),
&SCALAR_7);
        let base_rate = util_scalar.fixed_mul_ceil(e, &i128(config.r_two), &SCALAR_7)
            + i128(config.r_one)
            + i128(config.r_base);

        cur_ir = base_rate.fixed_mul_ceil(e, &ir_mod, &SCALAR_7);
```

Description of the potential exploit:

1. A pool is deployed with config.util = 0.95
2. The current utilization of the reserve is also currently 0.95
3. When accrual is calculated, the code calculates the current interest rate using r_one and r_base (if) instead of using r_two (else). This results in a different interest rate than expected.

We demonstrate the problem also using the Certora Prover on a simplified version of the code which preserves the control flow. See Certora Prover run link here.

Recommendation: Add the missing check.

Customer's Response: Fixed in PR#46.

Fix Review: Fix confirmed (now capped at 0.9).

L-03. Token to shares ratio can be 1:1

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
	Status: Acknowledged	

Description: A common practice in lending pools is to mint a minimum balance of tokens to the burn address or the pool creator. In essence, this means when new shares are minted, no one can ever own the 100% of the pool shares. Giving and allowing users to own as many shares as there are tokens, means that there are high-liquidity attacks that attackers can use, or worse, the calculation of the conversion functions returning the wrong value.

This type of attack has been seen in the wild in rate manipulation exploits, including that of Balancer, where it mints a BPT (Balancer Pool token) in exchange for tokens provided for supply. In their post-mortem after their hack, they make a similar assumption that a 1:1 rate is only possible on pool initialization (except for Blend, it's for first deposit); and that once a pool has liquidity, the ratio cannot be equal to 1:1 again.

```
Unset
Because all their BPT is pre-minted, and they do not allow conventional joins, Linear Pools use an unusual method for initialization: a setup trade, where the main token is swapped in for some of the pre-minted BPT.

The rate would be undefined during this first swap, when the circulating supply is zero, so in place of the usual rate calculation, we simply return BPT for main 1-to-1 (effectively bootstrapping the initial rate to 1). Needless to say, this should only happen once.

Most pool types have a MINIMUM_BPT amount that is burned on initialization, guaranteeing that even if all LPs exit, the supply cannot return to zero. Given this guarantee, checking for a total supply of zero is an easy way to determine whether the pool is being initialized (and should have a rate of 1).

Source: Balancer postmortem on rate manipulation exploit (asset lost estimated at ~$1.2M).
```

While it is true that the issue does not seem to be directly exploitable in Blend V2, best practice is to take measures to prevent it, in order to reduce the attack surface on the protocol.

Customer's Response: Acknowledged, but won't fix, as we don't think this risk applies to Blend v2 backstop shares.

Informational Severity Issues

I-01. Missing or incorrect documentation

Description: The following items are missing documentation or have incorrect/outdated information:

Token reserve indexes

Usage of index*2+1, index*2 should be documented throughout the codebase. From discussing with the Blend team, it appears that config.index is just the order in which reserves are added to the pool, and the order of the reserve contract address entry in the reserve_list. Emissions can be applied to either/both b_tokens or d_tokens for a reserve, based on if the pool is sending emissions to suppliers or borrowers (hence the x2). Thus, emissions are applied to an reserve_token_id, such that:

```
Unset
d_token_id = reserve_index * 2

b_token_id = reserve_index * 2 + 1
```

So if XLM is reserve 0 and USDC is reserve 1:

```
Unset
XLM d_token -> 0

XLM b_token -> 1

USDC d_token -> 2

USDC b_token ->  3
```

Customer's Response: Fixed in PR#46.

Fix Review: Fix confirmed.

Reward zones

The whitepaper specifies that the number of pools that can be in the reward zone will change over time, such that a new pool can be added every 97 days.

Reward Zone

For a lending pool to receive emissions, it must be part of the reward zone. The reward zone is a subset of pools with the largest backstop modules; at protocol release, it will include 10 pools and add 1 pool every 97 days. A pool in the reward zone can be replaced at any point if a pool outside the reward zone has more backstop deposits. To be added to the reward zone, pools must have the status Active.

In V2, there can only be a maximum of 50 pools in a reward zone, regardless of how much time has passed since genesis.

Customer's Response: Acknowledged, will be fixed in v2 docs update.

Max Positions

The max_positions field requires additional documentation for how it is expected to change over time, and additional analysis on when these max_positions are changed. The number is set on creation, and can be changed by an admin at any time.

When asking Blend, they clarified that the max_positions field depended on Soroban's resource limits, and should hypothetically only go up to support additional positions. If this is the case, an additional check should be added to ensure that this field can only increase.

If decreasing is warranted, this requires additional testing and documentation to test the proper behaviour of specific functions once max positions decreases.

Customer's Response: Fixed. Test added in PR#49.
Fix Review: Fix confirmed.

Flash loan implementation

The flash loan implementation implements an execute_submit with an implicit borrow function, without the explicit requirement to return the funds that were initially borrowed, so long as the health factor is greater than the stated minimum.

This differs from general flash loan implementations that requires the asset, amount + interest be returned post-flash loan, and requires in-depth documentation for use cases.

Customer's Response: Acknowledged, targeted for v2 docs.

Different token implementations

Token implementations such as tokens with fees or rebasing tokens can potentially be problematic for the accounting in the system. The support and impact of adding these tokens to the codebase should be clearly documented, so users know not to interact with pools with non-standard tokens.

Customer's Response: Acknowledged, targeted for both v1 and v2 docs.

The function apply_repay

In L.#393 of actions.rs , we should change "b_tokens_burnt" to "d_tokens_burnt".

Customer's Response: Fixed in PR#49.
Fix Review: Fix confirmed.

The function `set_reserve`

The inline documentation found in L.#68 of `contract.rs`, states that the function `set_reserve` should only be called by an admin. However this is not true and (as correctly implemented in the code itself) the function `set_reserve` can be invoked by anyone, since access is controlled by `queue_set_reserve`, where the new reserve config is provided by the admin. In addition, the function `set_reserve` is also missing a TTL extension.

Customer's Response: Fixed in PR#49.
Fix Review: Fix confirmed.

I-02. Code quality and best practices

Description: Some general suggestions -

- 1_0000100 should be made constant, to improve the readability of the code.. Similarly, for the thresholds 1_1500000 and 1_0300000 used in the user liquidation auction.
- setters like set_user_emissions returning a value and that too different from the set value is confusing.
- execute_submit_with_flash_loan and execute_submit could benefit from some code reuse since some of the code is common in both.

Customer's response: Acknowledged. Some code refactored where these numbers are not re-used across different files, so we kept the constants inline.

I-03. Spelling mistakes

Description:

- In L.#11 of health_factor.rs replace "demoninated" with "denominated".
- In L.#15 of health_factor.rs replace "demoninated" with "denominated".
- In L.#43 of backstop_interest_auction.rs replace "the the" with "the".
- In L.#121 of submit.rs replace "involed" with "involved".

Customer's Response: Fixed in PR#49.

Fix Review: Fix confirmed.

Appendix A. Architectural Design Recommendations

The following appendix contains design recommendations for both the current V2 implementation, as well as a larger refactor recommendation for V3. During this review, a significant amount of time was spent understanding the codebase. These recommendations intend to simplify the codebase.

Recommendations for Blend V2

- **Simplify the gulp functions.** There are gulp_* or do_gulp_* functions in backstop/contracts.rs, backstop/src/emissions/manager.rs, pool/src/contract.rs that call across different functions - it becomes hard to read, and hard to understand the expected execution flow.
- **Pool statuses have different threshold bounds.** Pool statuses 1, 3, 5 are used for backstop threshold; and 0, 2, 4, 6 for admin states. Each of these states require a different q4w threshold, and should be simplified. While documentation exists on how these states are expected to change, this results in complex invariants.
- **Consistent terminology between gulping emissions and accruing interest.** The multiple gulp functions and accrue interest functions makes the codebase hard to read. It also becomes harder to tell the impact on system variables.
- **Simplifying and documenting load() functions.** The current load functions tend to change ir_mod, d_rate, and gulp. Trying to track how these variables change is significantly more complicated, as well as determining a consistent pattern when these variables are changed.
- **The backstop has separate deposit/withdrawal files, whereas the pool uses apply_* functions as entrypoints through built actions.** The codebase should either choose to implement one variation or the other, to ensure that entrypoints are explicitly shown to users, and to allow developers a quicker understanding of how the system calls intend to work.

The following provide additional holistic recommendations for the codebase to increase readability:

- **Weigh the advantages and disadvantages of having a centralized reserve in the backstop vs individual reserves in pools.** A centralized reserve may be easier for the backstop, however it means a vulnerability in the backstop puts the entire reserve, even of other pools, at risk.
- **Reuse code.** Wherever possible, ensure that code is not duplicated across multiple functions. This makes analyzing bugs harder, as well as increases the attack surface.

- **Always use constants over hard-coded variables.** This ensures that if there is a need to change limits and constants, that the change can be made efficiently by adjusting one variable, instead of needing to refactor across multiple functions in the codebase. Performing the latter operation can lead to mistakes, where not every instance is changed, and result in future security vulnerabilities.
- **Consistency in getters (returning variables) and setters (setting arguments).** Avoid having getters that also change state, and having setters that return arguments - instead, add a separate function call in the caller of the setter function to execute additional calculation. Returning variables from setter functions makes the codebase much harder to read.
- **Document return value type, especially for scaled values.** Some functions return values of tokens as provided, while other functions return a number scaled by one of the scalar constants. Problematic issues can arise when one function makes the assumption that the return value is expected to be scaled by a certain amount, and it is not. Thus, each function that handles numbered values for calculation should specify what factor the arguments and return value is in.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

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.