code4rena

# Blend

## Smart Contract Security Assessment

Audit dates: Feb 24 — Mar 17, 2025

# Overview

## About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Blend V2 smart contract system, for the team at Script3. The audit took place from February 24 to March 17, 2025.

Following the C4 audit, 6 wardens (0x007, oakcobalt, Testerbot, rscodes and a_kalout and ali_shehab of team 0xAlix2) reviewed the mitigations implemented by the Script3 team; the mitigation review report is appended below the audit report.

Final report assembled by Code4rena.

# Summary

The C4 analysis yielded an aggregated total of 21 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 18 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 6 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Script3 team.

# Scope

The code under review can be found within the C4 Blend V2 Audit + Certora Formal Verification repository, and is composed of 62 files written in the Rust programming language and includes 27,099 lines of Rust code.

# Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

## High Risk Findings (3)

### [H-01] A reserve's `d_supply` is incorrectly updated and stored after flash loan execution

*Submitted by alexxander, also found by 0x007, 0xAlix2, aldarion, audithare, carrotsmuggler, klau5, mahdikarimi, oakcobalt, rapid, rscodes, Testerbot, and Tricko*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/submit.rs#L86

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/submit.rs#L101

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/actions.rs#L187

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/actions.rs#L412

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/actions.rs#L417

#### Finding description and impact

Executing `contract.flash_loan(...)` will subsequently call `submit.execute_submit_with_flash_loan(...)` where the `Pool` struct is loaded with `Pool.config` assigned to the pool's configuration and all other `Pool` fields are empty: `let mut pool = Pool::load(e)`.

After that, the Reserve of the asset that will be borrowed as a flash loan is loaded - `let mut reserve = pool.load_reserve(e, &flash_loan.asset, true)`. The function `pool.load_reserve(...)` caches the Address of the asset in the array `Pool.reserves_to_store` and loads the Reserve from storage since the Reserve is not yet present in the Map `Pool.reserves`.

The execute function then adds the flash loan amount as a liability for the user through `from_state.add_liabilities(e, &mut reserve, d_tokens_minted)` which will update the `d_supply` of the Reserve. However, the execute function has not cached the Reserve in `Pool.reserves` through the function `pool.cache_reserve(...)`.

```
pub fn execute_submit_with_flash_loan(...) -> Positions {
    ...
    let mut pool = Pool::load(e);
    let mut from_state = User::load(e, from);
    ...
    {
        let mut reserve = pool.load_reserve(e, &flash_loan.asset, true);
        let d_tokens_minted = reserve.to_d_token_up(e,
flash_loan.amount);
        from_state.add_liabilities(e, &mut reserve, d_tokens_minted);
        ...
    }
    let actions = build_actions_from_request(e, &mut pool, &mut
from_state, requests);
    ...
    pool.store_cached_reserves(e);
    ...
}
```

The execute function then continues to process the rest of the user's requests by calling `build_actions_from_request(...)`. However, if the asset used for the flash loan is also used in the requests, such as a Repay Request to payback the flash loaned amount, `build_actions_from_request(...)` will call; for example, `apply_repay(...)` which will attempt to load the Reserve for the same token that was borrowed by the user.

Since the flash loan asset's Reserve wasn't originally cached with `pool.cache_reserve()`, the Reserve will again be loaded from storage where the `d_supply` is the value before the flash loan liability was added. The `apply_repay()` function will then subtract the repaid flash loan amount from the stale `d_supply` (before the flash loan liability was accounted for) and will call `pool.cache_reserve()` to cache the reserve in `Pool.reserves`. Finally, after `build_actions_from_request(...)` has finished processing requests, the reserves cached

in `Pool.reserves` are saved in storage through `pool.store_cached_reserves(e)` and the flash loaned asset's Reserve will be stored with an incorrect `d_supply`.

```
fn apply_repay(...) -> (i128, i128) {
    let mut reserve = pool.load_reserve(e, &request.address, true);
    ...
    if d_tokens_burnt > cur_d_tokens {
        ...
        user.remove_liabilities(e, &mut reserve, cur_d_tokens);
        pool.cache_reserve(reserve);
        ...
    } else {
        ...
        user.remove_liabilities(e, &mut reserve, d_tokens_burnt);
        pool.cache_reserve(reserve);
        ...
    }
}
```

A summary through an example:

- Asset X has a Reserve with `d_supply = 500` and a 1:1 rate of X tokens to `d_tokens`
- User calls `contract.flash_loan(...)` with a `Flash_Loan.asset = X`, `Flash_Loan.amount = 250` and a Repay Request where `Request.asset = X` and `Request.amount = 250`
- X Reserve is loaded with `d_supply = 500`
- Call to `from_state.add_liabilities(e, &mut reserve, d_tokens_minted)` sets `d_supply = 750`
- X Reserve is not cached or saved in storage
- `build_actions_from_request(...)` processes the Repay Request
- `apply_repay(...)` calls `pool.load_reserve(...)`
- The X Reserve is again loaded from storage with `d_supply = 500`
- `apply_repay(...)` calls `user.remove_liabilities(...)` which sets `d_supply = 250`
- `apply_repay(...)` saves in cache X Reserve with `d_supply = 250`
- Finally, `pool.store_cached_reserves(e)` is called and X Reserve is saved in storage with `d_supply = 250`

### Impact

A core invariant of the system is violated where for a Reserve, the sum of all user's liabilities must be equal to `d_supply`. There are numerous places where `d_supply` is used within the protocol and when updated incorrectly will cause erroneous calculations:

- The function `reserve.utilization()` uses `d_supply` to determine how much of the Reserve's liquidity can be given as a loan. An incorrect decrease in `d_supply` will allow borrowing assets even if the maximum utilization ratio is met.
- The utilization of the reserve is also used in calculating the accrual fees of the Reserve in `reserve.load(...)` which downstream calls `interest.calc_accrual(...)`. An incorrect update of the accrued interest can be of benefit to some users and a drawback to others.
- Since `d_supply` is an `i128` value, a `d_supply` that is lower than the sum of all user's liabilities in the Reserve can cause `d_supply` to become negative when `user.remove_liabilities(...)` is executed. As per the assumptions of the protocol, `d_supply` should never be a negative number.
- The distribution of BLEND token emissions will be incorrect when `d_supply` incorrectly decreases, allowing users to illegally claim more BLEND tokens than they are owed, damaging other Reserve's emission rewards and denying other users from claiming rewards.

A short example of an attack idea where a malicious user claims illegally BLEND token emissions:

- Assume Reserve for token X with `d_supply = N`.
- Assume a malicious user has some existing liability, participating in the `d_supply` for Reserve X .
- Malicious user calls `contract.flash_loan(...)` and exploits to reduce the `d_supply = 1` with a Flash Loan amount of `N-1` and a Repay Request of `N-1`.
- When time passes, `distributor.update_emission_data(...)` will update the index (reward per token) for the Reserve X according to `d_supply = 1`; i.e. inflating the reward per token (there are still other users that have liabilities but `d_supply` does not reflect that).
- Malicious user calls `contract.claim()` for Reserve X where he will claim an inflated amount of accrued rewards since the index (reward per token) is inflated.
- Subsequent accounts of the malicious user or regular users can continue claiming their rewards for Reserve X with an inflated index, therefore, stealing other Reserves rewards.

### Proof of Concept

- In `/blend-contracts-v2/pool/src/pool/submit.rs`
- Apply the modifications below to the the test `test_submit_with_flash_loan_process_flash_loan_first()`
- Change directory to `/blend-contracts-v2/pool`
- Run with `cargo test pool::submit::tests::test_submit_with_flash_loan_process_flash_loan_first -- --nocapture --exact`
- Inspect the `@` audit tags in the test and the log output in the console

▶ Details

### Recommended mitigation steps

In `submit.execute_submit_with_flash_loan()`, use `pool.cache_reserve()` to cache the Reserve of the Flash Loan asset after adding the flash loan amount as a liability for the user.

**markus_pl10 (Script3) confirmed**

**mootz12 (Script3) commented:**

Fixed to add `cache_reserve()` call after applying state changes.

**Blend mitigated:**

Commit e4ed914 to clean up flash loans implementation.

**Status:** Mitigation confirmed. Full details in reports from 0xAlix2, Testerbot, 0x007, oakcobalt and rscodes.

---

## [H-02] User can steal other users' emissions due to vulnerable claim implementation

*Submitted by oakcobalt, also found by 0xabhay, cu5t0mpeo, mahdikarimi, and Testerbot*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/contract.rs#L1

### Finding description

`backstop::emissions::execute_claim` is missing `update_emissions` for `to` address which allows a user stealing other users' emissions.

### Proof of Concept

When a user deposit to backstop, `update_emissions` needs to be called atomically before `user_balance` update to ensure user emissions are initialized correctly; e.g. if user has zero share, their emission data should be intiralized first to current index.

However, in `execute_claim`, when `from ≠ to`, the exchanged backstop LPs are deposited to `to` address but `update_emissions` is missing. This means `to`'s balance is updated without syncing/initializing their emission data first before the balance change. This introduces a state synchronization conflict. `to`'s emissions will be inflated.

```
pub fn execute_claim(e: &Env, from: &Address, pool_addresses:
&Vec<Address>, to: &Address) -> i128 {
```

```
...
            for pool_id in pool_addresses.iter() {
                        let claim_amount =
claims.get(pool_id.clone()).unwrap();
            let deposit_amount = lp_tokens_out
                .fixed_mul_floor(claim_amount, claimed)
                .unwrap();
            let mut pool_balance = storage::get_pool_balance(e,
&pool_id);
            let mut user_balance = storage::get_user_balance(e, &pool_id,
to);

            // Deposit LP tokens into pool backstop
            let to_mint = pool_balance.convert_to_shares(deposit_amount);
            pool_balance.deposit(deposit_amount, to_mint);
            //@audit `to`'s balance is updated without update_emissions.
`to`'s emission data is not synced/ initialized.
|>          user_balance.add_shares(to_mint);

            storage::set_pool_balance(e, &pool_id, &pool_balance);
            storage::set_user_balance(e, &pool_id, to, &user_balance);

            BackstopEvents::deposit(e, pool_id, to.clone(),
deposit_amount, to_mint);
        }
        ...
```

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/emissions/claim.rs#L66

Flows: `backstop::claim -> emissions::execute_claim`

One exploit scenarios is a user can `execute_claim` into an address of their control and such address has no user balance.

### Coded PoC

Suppose userA and userB have deposited equal share in backstop module. userA controls addressC which has no user emission data.

1. userA claim emissions and deposit to addressC.
2. userA immediately claim emissions again through addressC. addressC claimed historical emissions even though it has 0 share prior to userA's first claim.
3. userB tries to claim their emissions. tx revert due to insufficient funds. userA stole emissions from userB successfully.

See added unit test `test_user_steal_emissions()` in `backstop/src/emissions/claim.rs`. Run test: `cargo test test_user_steal_emissions`.

▶ Details

Test results:

```
     Running unittests src/lib.rs (target/debug/deps/backstop-
70626b884282a67b)

running 1 test
test emissions::claim::tests::test_user_steal_emissions - should panic
... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 103 filtered
out; finished in 0.24s
```

### Impact

When `from != to`, `to` can claim part of other user's emissions. A user can steal emissions by inputting a `to` address new to the backstop module. The attack can be repeated with multiple user controlled addresses.

### Recommended mitigation steps

In execute_claim, add a logic when `from ≠ to` invoke `update_emissions` for `to` before adding shares to `to`.

**markus_pl10 (Script3) confirmed**

**Comments from the Script3 team:**

While this finding was excluded from the scope of the mitigation review, it was addressed here by removing the `to` address, which ensures that the exploit is no longer possible.

---

## [H-03] Utilization ratio can exceed 100% due to missing validation in withdrawal functions

*Submitted by 0x007*

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/pool/src/pool/actions.rs#L382

### Finding description

The protocol implements a maximum utilization ratio check via the `require_utilization_below_max` function, which ensures that `utilization = total_liabilities / total_supply` remains below a specific threshold. However, this validation is only enforced in the `apply_borrow` function when increasing liabilities, and is critically missing from the `apply_withdraw` and `apply_withdraw_collateral` functions.

Since withdrawals reduce the `total_supply` denominator in the `utilization` formula, large withdrawals can cause the utilization ratio to increase beyond the intended maximum threshold. These seems intentional but there are downsides when utilization exceeds 100%. The extra amounts can come from `backstop_interest` or donations.

### Impact

1. **Unbounded Interest Rates:** The protocol's interest rate models uses utilization ratio as a key input. When utilization exceeds 100%, the interest rate calculation can produce extreme values.
2. **Market Instability:**
    - Extended periods with utilization ratio above 100% can lead to liquidations, bad debt accumulation and potential protocol insolvency.
    - `backstop_interest_auction` might not work because the interest has been lent out

### Proof of Concept

This was slightly modified from
[test_build_actions_from_request_withdraw_allows_over_max_util]

```
#[test]
fn test_build_actions_from_request_withdraw_allows_over_100_util() {
    let e = Env::default();
    e.mock_all_auths();

    let bombadil = Address::generate(&e);
    let samwise = Address::generate(&e);
    let pool = testutils::create_pool(&e);

    let (underlying, _) = testutils::create_token_contract(&e,
&bombadil);
    let (mut reserve_config, mut reserve_data) =
testutils::default_reserve_meta();
    reserve_config.max_util = 0_9000000;
    reserve_data.b_supply = 100_0000000;
    reserve_data.d_supply = 89_0000000;
```

```
    testutils::create_reserve(&e, &pool, &underlying, &reserve_config,
&reserve_data);

    e.ledger().set(LedgerInfo {
        timestamp: 600,
        protocol_version: 22,
        sequence_number: 1234,
        network_id: Default::default(),
        base_reserve: 10,
        min_temp_entry_ttl: 10,
        min_persistent_entry_ttl: 10,
        max_entry_ttl: 3110400,
    });
    let pool_config = PoolConfig {
        oracle: Address::generate(&e),
        min_collateral: 1_0000000,
        bstop_rate: 0_2000000,
        status: 0,
        max_positions: 2,
    };

    let user_positions = Positions {
        liabilities: map![&e],
        collateral: map![&e],
        supply: map![&e, (0, 20_0000000)],
    };
    e.as_contract(&pool, || {
        storage::set_pool_config(&e, &pool_config);
        storage::set_user_positions(&e, &samwise, &user_positions);

        let mut pool = Pool::load(&e);

        let requests = vec![
            &e,
            Request {
                request_type: RequestType::Withdraw as u32,
                address: underlying.clone(),
                // diff1 from: amount: 2_0000000,
                amount: 20_0000000,
            },
        ];
        let mut user = User::load(&e, &samwise);
        let actions = build_actions_from_request(&e, &mut pool, &mut
user, requests);

        assert_eq!(actions.check_health, false);
```

```
        let spender_transfer = actions.spender_transfer;
        let pool_transfer = actions.pool_transfer;
        assert_eq!(spender_transfer.len(), 0);
        assert_eq!(pool_transfer.len(), 1);
        // diff2 from: assert_eq!
(pool_transfer.get_unchecked(underlying.clone()), 2_0000000);
        assert_eq!(pool_transfer.get_unchecked(underlying.clone()),
20_0000000);

        let positions = user.positions.clone();
        assert_eq!(positions.liabilities.len(), 0);
        assert_eq!(positions.collateral.len(), 0);
        assert_eq!(positions.supply.len(), 1);
        // diff3 from: assert_eq!(user.get_supply(0), 18_0000111);
        assert_eq!(user.get_supply(0), 1110);

        // diff4: add utilization rate check
        let reserve = pool.load_reserve(&e, &underlying.clone(), false);
        assert_eq!(reserve.utilization(&e), 1_1125010);
    });
}
```

### Recommended Mitigation

Add the `require_utilization_below_max` validation to both the `apply_withdraw` and `apply_withdraw_collateral` functions. This ensures that all operations that could potentially increase the utilization ratio (either by increasing liabilities or decreasing supply) are properly validated against the maximum threshold. If you want to permit withdrawal beyond threshold, add a `require_utilization_below_100` for withdrawals.

**markus_pl10 (Script3) confirmed**

Blend mitigated:

PR 48 - Validate if util is below 100% when doing withdraw actions (see commit).

**Status:** Mitigation confirmed. Full details in reports from 0x007, 0xAlix2, Testerbot and oakcobalt.

---

## Medium Risk Findings (18)

### [M-01] Flash loans allow borrowing from frozen pools, bypassing security controls

*Submitted by Testerbot, also found by 0x007, aldarion, carrotsmuggler, oakcobalt, peakbolt, and YouCrossTheLineAlfie*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/submit.rs#L868-L895

### Finding description and impact

Changing a pool's status is crucial for risk management in Blend's lending protocol, serving as an automatic circuit breaker that responds to changing risk conditions. The three states (Active, On Ice, Frozen) are triggered based on backstop depositors' withdrawal behavior, as these depositors provide first-loss capital and are most sensitive to risk. When withdrawal queues reach certain thresholds, the protocol restricts operations accordingly, preventing further risk accumulation during uncertain market conditions or when the pool's health is deteriorating. Pool owners can also manually put the pool on-ice or even frozen it to respond to risks they observe, ensuring the protocol remains resilient against any issues.

The rules for these changes are outlined in the functions:

- `execute_set_pool_status()`, here.
- `execute_update_pool_status()`, here.

The pool status is checked before processing any operation in the pool, this can be observed in the `build_actions_from_request()` function, here. The validation happens in the `pool.rs` file under the `require_action_allowed()` function:

```
pub fn require_action_allowed(&self, e: &Env, action_type: u32) {
        // disable borrowing or auction cancellation for any non-active
pool and disable supplying for any frozen pool
        if (self.config.status > 1 && (action_type == 4 || action_type ==
9))
            || (self.config.status > 3 && (action_type == 2 ||
action_type == 0))
        {
            panic_with_error!(e, PoolError::InvalidPoolStatus);
        }
    }
```

If the pool is the frozen or even on-ice state, the protocol should panic.

However, there's a critical security vulnerability in the flash loan implementation. In the way the flash loans are implemented, a user is not obligated to return the borrowed assets, instead the user can keep the funds as a borrow as long as they have a healthy position after the flash loan. In this way, the flash loans can be used to make a simple borrow operation.

The issue arises because the flash loan implementation do not implement the pool and reserve status validations that are present in the normal borrowing flow in fuctions like `build_actions_from_request()` and `apply_borrow()` with the `pool.require_action_allowed()` and `reserve.require_action_allowed()` checks respectively.

### Proof of concept

1. Admin freezes the pool by setting the pool status to indicate it's frozen
    - The pool can get frozen permissionless also if the backstop withdrawals reach a certain threshold.
2. Normal borrowing attempts are rejected due to the validation check in `build_actions_from_request()`.
3. However, an attacker can still borrow assets by using the flash loan functionality.

Follow the next steps to reproduce the issue in a coded test:

In the `test-suites/tests/test_flash_loan.rs` file, paste the following test:

```
#[test]
fn test_flashloan_bypass_frozen_pool() {
    let fixture = create_fixture_with_data(true);
    let pool_fixture = &fixture.pools[0];
    let frodo = fixture.users[0].clone();

    let xlm = &fixture.tokens[TokenIndex::XLM];
    let xlm_address = xlm.address.clone();
    let stable = &fixture.tokens[TokenIndex::STABLE];
    let stable_address = stable.address.clone();

    let (receiver_address, _) = create_flashloan_receiver(&fixture.env);

    let samwise = Address::generate(&fixture.env);

    let approval_ledger = fixture.env.ledger().sequence() + 17280;

    xlm.mint(&samwise, &(100 * SCALAR_7));
    xlm.approve(
        &samwise,
        &pool_fixture.pool.address,
        &i128::MAX,
        &approval_ledger,
    );
    stable.mint(&samwise, &(100 * SCALAR_7));
    stable.approve(
```

```
        &samwise,
        &pool_fixture.pool.address,
        &i128::MAX,
        &approval_ledger,
    );

    let supply_collateral_request: Vec<Request> = vec![
        &fixture.env,
        Request {
            request_type: RequestType::SupplyCollateral as u32,
            address: stable_address.clone(),
            amount: 50 * SCALAR_7,
        },
    ];

    pool_fixture.pool.submit(&samwise, &samwise, &samwise,
&supply_collateral_request);

    let flash_loan = FlashLoan {
        contract: receiver_address.clone(),
        asset: xlm_address.clone(),
        amount: 1_000 * SCALAR_7,
    };

    // No requests.
    let actions_request: Vec<Request> = vec![
        &fixture.env,
    ];

    // Pool is frozen.
    pool_fixture.pool.set_status(&4);
    let pool_status = pool_fixture.pool.get_config();
    assert_eq!(pool_status.status, 4);

    // However flash loan will go through.
    pool_fixture
        .pool
        .flash_loan(&samwise, &flash_loan, &actions_request);
}
```

Run the test with `cargo test --test test_flashloan -- --nocapture -- test_flashloan_bypass_frozen_pool`.

### Recommended mitigation steps

Add the validations `pool.require_action_allowed()` and `reserve.require_action_allowed()` in the flash loan implementation.

**markus_pl10 (Script3) confirmed**

**Blend mitigated:**

Commit e4ed914 to clean up flash loans implementation.

**Status:** Initial fix resolved the issue described in Testerbot's submission S-276, but did not resolve the issue described in duplicate submission S-308 by peakbolt. Please refer to the Mitigation Review section of this report for additional details.

---

## [M-02] Invalid utilization ratio check, blocking users from submitting a flash loan

*Submitted by OxAlix2*

Each asset in a pool has a utilization ratio, which refers to the percentage of the asset's deposits that are currently being borrowed. A pool admin can set the max utilization ratio for each asset, which shouldn't be bypassed; this is referred to as `max_util`.

The asset's utilization ratio is checked whenever a borrow is made:

```
    fn apply_borrow(
        e: &Env,
        actions: &mut Actions,
        pool: &mut Pool,
        user: &mut User,
        request: &Request,
    ) -> i128 {
        let mut reserve = pool.load_reserve(e, &request.address, true);
        reserve.require_action_allowed(e, request.request_type);
        let d_tokens_minted = reserve.to_d_token_up(e, request.amount);
        user.add_liabilities(e, &mut reserve, d_tokens_minted);
@>      reserve.require_utilization_below_max(e);
        actions.add_for_pool_transfer(&reserve.asset, request.amount);
        actions.do_check_health();
        pool.cache_reserve(reserve);
        d_tokens_minted
    }
```

On the other hand, each pool allows users to submit a flash loan, which consists of taking a loan, doing some actions, and possibly paying it back in the same transaction.

However, when submitting a flash loan, and taking that loan (1@), the utilization ratio is checked immediately after it (2@).

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/pool/src/pool/submit.rs#L89

```
    pub fn execute_submit_with_flash_loan(
        e: &Env,
        from: &Address,
        flash_loan: FlashLoan,
        requests: Vec<Request>,
    ) -> Positions {
        if from == &e.current_contract_address() {
            panic_with_error!(e, &PoolError::BadRequest);
        }
        let mut pool = Pool::load(e);
        let mut from_state = User::load(e, from);

        let prev_positions_count =
from_state.positions.effective_count();

        // note: we add the flash loan liabilities before processing the
other
        // requests.
        {
            let mut reserve = pool.load_reserve(e, &flash_loan.asset,
true);
            let d_tokens_minted = reserve.to_d_token_up(e,
flash_loan.amount);
1@>         from_state.add_liabilities(e, &mut reserve, d_tokens_minted);
2@>         reserve.require_utilization_below_max(e);

            PoolEvents::flash_loan(
                e,
                flash_loan.asset.clone(),
                from.clone(),
                flash_loan.contract.clone(),
                flash_loan.amount,
                d_tokens_minted,
            );
        }

        // ... snip ...
    }
```

This is wrong, as it doesn't give the user a chance to execute the flash loan, then repay it, and would revert. For example, A user submits a flash loan with 1k USDC. With that new debt, the utilization ratio exceeds the max utilization ratio. However, the user wants to take that loan, do some actions (we don't really care), and wants to repay it after those actions. As a result, the end state after the transaction ends, the utilization ratio will be below the max; i.e., healthy.

With the current place of `require_utilization_below_max`, it blocks that user from executing that legit flashloan scenario.

### Proof of Concept

Add the following test in `blend-contracts-v2/pool/src/pool/submit.rs`:

```
#[test]
#[should_panic(expected = "Error(Contract, #1207)")]
fn test_submit_with_flash_loan_wrong_max_util() {
    let e = Env::default();
    e.cost_estimate().budget().reset_unlimited();
    e.mock_all_auths_allowing_non_root_auth();

    e.ledger().set(LedgerInfo {
        timestamp: 600,
        protocol_version: 22,
        sequence_number: 1234,
        network_id: Default::default(),
        base_reserve: 10,
        min_temp_entry_ttl: 10,
        min_persistent_entry_ttl: 10,
        max_entry_ttl: 3110400,
    });

    let bombadil = Address::generate(&e);
    let samwise = Address::generate(&e);
    let pool = testutils::create_pool(&e);
    let (oracle, oracle_client) = testutils::create_mock_oracle(&e);

    let (flash_loan_receiver, _) =
testutils::create_flashloan_receiver(&e);

    let (underlying_0, underlying_0_client) =
testutils::create_token_contract(&e, &bombadil);
    let (mut reserve_config, mut reserve_data) =
testutils::default_reserve_meta();
    reserve_config.max_util = 9500000;
    reserve_data.b_supply = 100_0000000;
```

```
    reserve_data.d_supply = 50_0000000;
    testutils::create_reserve(&e, &pool, &underlying_0, &reserve_config,
&reserve_data);

    let (underlying_1, underlying_1_client) =
testutils::create_token_contract(&e, &bombadil);
    let (reserve_config, reserve_data) =
testutils::default_reserve_meta();
    testutils::create_reserve(&e, &pool, &underlying_1, &reserve_config,
&reserve_data);

    oracle_client.set_data(
        &bombadil,
        &Asset::Other(Symbol::new(&e, "USD")),
        &vec![
            &e,
            Asset::Stellar(underlying_0.clone()),
            Asset::Stellar(underlying_1.clone()),
        ],
        &7,
        &300,
    );
    oracle_client.set_price_stable(&vec![&e, 1_0000000, 5_0000000]);

    e.as_contract(&pool, || {
        storage::set_pool_config(
            &e,
            &PoolConfig {
                oracle,
                min_collateral: 1_0000000,
                bstop_rate: 0_1000000,
                status: 0,
                max_positions: 4,
            },
        );

        underlying_1_client.mint(&samwise, &50_0000000);
        underlying_1_client.approve(&samwise, &pool, &100_0000000,
&10000);

        underlying_0_client.mint(&samwise, &46_0000000);
        underlying_0_client.approve(&samwise, &pool, &46_0000000,
&10000);

        // User takes a flash loan of 46_0000000, then supplies
50_0000000 of underlying_1, and finally repays the flash loan
```

```
        execute_submit_with_flash_loan(
            &e,
            &samwise,
            FlashLoan {
                contract: flash_loan_receiver,
                asset: underlying_0.clone(),
                amount: 46_0000000,
            },
            vec![
                &e,
                Request {
                    request_type: RequestType::SupplyCollateral as u32,
                    address: underlying_1,
                    amount: 50_0000000,
                },
                Request {
                    request_type: RequestType::Repay as u32,
                    address: underlying_0.clone(),
                    amount: 46_0000000,
                },
            ],
        );
    });
}
```

### Recommended mitigation steps

```
    pub fn execute_submit_with_flash_loan(
        e: &Env,
        from: &Address,
        flash_loan: FlashLoan,
        requests: Vec<Request>,
    ) -> Positions {
        // ... snip ...

        // note: we add the flash loan liabilities before processing the
other
        // requests.
        {
            let mut reserve = pool.load_reserve(e, &flash_loan.asset,
true);
            let d_tokens_minted = reserve.to_d_token_up(e,
flash_loan.amount);
            from_state.add_liabilities(e, &mut reserve, d_tokens_minted);
-           reserve.require_utilization_below_max(e);
```

```
            PoolEvents::flash_loan(
                e,
                flash_loan.asset.clone(),
                from.clone(),
                flash_loan.contract.clone(),
                flash_loan.amount,
                d_tokens_minted,
            );
        }

        // ... snip ...

        // store updated info to ledger
        pool.store_cached_reserves(e);
        from_state.store(e);

+       pool.load_reserve(e, &flash_loan.asset,
false).require_utilization_below_max(e);

        from_state.positions
    }
```

**markus_pl10 (Script3) confirmed**

mootz12 (Script3) commented:

Validated this is an issue. It's more of an implementation detail rather than an finding, as no funds or functionality is at risk.

Fixed to ensure flash loan is legal by checking flash loan under 100% util, then also check max util of asset during validation.

Blend mitigated:

Commit f35271b to clean up utilization checks.

**Status:** Mitigation confirmed. Full details in reports from 0xAlix2, 0x007, Testerbot and oakcobalt.

---

## [M-03] If a withdrawal executes after a bad debt auction gets created, it could cause the auction to be stuck and further bad debt auctions can't be created

*Submitted by rscodes, also found by 0xAlix2, attentioniayn, and rapid*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/auctions/bad_debt_auction.rs#L15-L139

### Finding description and impact

In a bad debt auction:

- Bid token - the `dtokens` (debt) the auction winner will take on.
- Lot token - The backstop token the pool will transfer to the auction winner.

From block 0 to 200, the amount of lot tokens given to winner scales **upwards** and peaks at block 200.

From block 200 to infinity, the **amount of lot tokens given to winner stays the same at the peak value.**

So the first key observation is that the lot tokens can only increase or stay the same (and throughout the auction, will **never** decrease).

The second key observation is that there is no option to remove a bad debt auction. So if a bad debt auction can't be filled then it'll just be stuck there. It is worth noting that if there is an existing bad debt auction, then `create_bad_debt_auction_data` will **not allow** a new bad debt auction. (Hence, resulting in the whole bad debt auction mechanism to be stuck)

- The only auction that has a function to remove it is the user liquidation auction. For bad debt auctions, only after filling it and a winner is chosen, then it will be removed.

The third observation is this line in `create_bad_debt_auction_data`:

- The Line: `lot_amount = pool_backstop_data.tokens.min(lot_amount)`
- Putting this in place was with the intention to ensure that the pool has sufficient backstop tokens so that the bad debt auction can be filled and wont be stuck.

However, combining the 3 observations, we can derive a scenario where malicious users can brick new bad debt auctions.

### Sequence of Attack

1. Suppose the backstop currently has `50,000e7` tokens.
2. `Attacker` initiates withdrawal of their `9,000e7` tokens. (And a one week lock starts)
3. Some time later:
    1. A bad debt auction is created. (Looking at the third observation, at this point of time, the pool still has `50,000e7` tokens so the min capping does not affect this).
    2. Let's say the optimal time to fill the auction is at block B, with users taking on X amount of debt in exchange for `42,000e7` tokens.
    3. So `victim` attempts to fill the auction at block B, however `Attacker` frontruns with `withdraw`. (and now the pool is left with `41_000e7` tokens).

4. Now, the pool does not have enough tokens and the attempt to fill the auction panics and reverts.

As explained in the first observation, since lot token amounts can only increase/stay the same, even if the victim continues waiting, this **bad debt auction can't be filled**

Now, leveraging on the second observation, the existing bad debt auction is stuck there, preventing new bad debt auctions from being created.

### Impact and likelihood

Since **new bad debt auctions can't be created, it causes certain losses for the lenders.** (Plus, this current bad debt isn't socialized properly as well and lenders take the damage for it eventually).

This is almost certain to happen to pools with lower backstop supply. However, the likelihood shouldn't be downgraded because of this as even some pools in uniswap have low supply. Furthermore, the low supply part is regarding the backstop pool, which is very likely to have less stakers than the main lender pool.

### Proof of Concept

Go to `bad_debt_auction.rs` and paste the PoC in the mod tests struct:

▶ Details

Run `cargo test test_withdrawal_during_auction` and we can see that it panics during `fill_bad_debt_auction`.

### Recommendation

In the `fill_bad_debt_auction` function, in the `backstop_client.draw` function, cap the `lot_amount` drawn to the current balance of the backstop pool.

Capping the lot token does not cause direct loss for the auction fillers as well because the bid token (which is the debt they have to take on in exchange) goes down. If they feel it's not worth it now, they can always wait and fill it a few blocks later where they take on less debt.

**markus_pl10 (Script3) confirmed**

Blend mitigated:

PR 48 to block `backstop::withdraw` calls if the backstop currently holds bad debt.

**Status:** Mitigation confirmed. Full details in reports from 0x007, Testerbot, 0xAlix2, oakcobalt and rscodes.

---

## [M-04] Users can create overpriced bad debt and interest auctions by providing duplicate reserves

*Submitted by 0xAlix2, also found by 0x007, 0xadrii, 0xtheauditor, aldarion, klau5, oakcobalt, slylandro_star and Testerbot*

When some interest/credit accumulates for a certain reserve, users can create an auction for that credit, in return for some backstop tokens that would be donated to the backstop pool, this could be done by calling `create_interest_auction_data`. This function checks the available credit for the provided reserves and computes the value of the whole credit in USDC, the amount of backstop tokens; i.e., the bid is calculated as a percentage of the USDC value.

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/pool/src/auctions/backstop_interest_auction.rs#L76-L84

```
let backstop_token_value_base =
    (pool_backstop_data
        .usdc
        .fixed_mul_floor(e, &oracle_scalar, &SCALAR_7)
        * 5)
    .fixed_div_floor(e, &pool_backstop_data.tokens, &SCALAR_7);
let bid_amount = interest_value
    .fixed_mul_floor(e, &1_2000000, &SCALAR_7)
    .fixed_div_floor(e, &backstop_token_value_base, &SCALAR_7);
```

The issue is that the protocol is looping through the provided assets, grabbing the value, and scaling it to USDC. However, the issue is that it doesn't check for duplicates.

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/pool/src/auctions/backstop_interest_auction.rs#L41-L57

```
// validate and create lot auction data
let mut interest_value = 0; // expressed in the oracle's decimals
for lot_asset in lot {
    // don't store updated reserve data back to ledger. This will occur
on the the auction's fill.
    // `load_reserve` will panic if the reserve does not exist
    let reserve = pool.load_reserve(e, &lot_asset, false);
    if reserve.data.backstop_credit > 0 {
        let asset_to_base = pool.load_price(e, &reserve.asset);
        interest_value += i128(asset_to_base).fixed_mul_floor(
            e,
            &reserve.data.backstop_credit,
            &reserve.scalar,
        );
        auction_data
```

```
                .lot
                .set(reserve.asset, reserve.data.backstop_credit);
        }
    }
```

This allows **anyone** to create an interest auction with a very low amount of credit (lot) for a very high amount of backstop tokens (bid), easily bypassing the minimum interest value check here.

For example, if a reserve has a credit that is worth `$100`, it could be passed 10 times in the `lot` and create an interest auction that is worth `$1000` of credit.

**NB: This issue also exists when creating a bad debt auction, here.**

### Proof of Concept

Add the following test in `blend-contracts-v2/pool/src/auctions/backstop_interest_auction.rs`:

▶ Details

### Recommended mitigation steps

Check and don't allow duplicate reserves when looping over the provided reserves in both `create_interest_auction_data` and `create_bad_debt_auction_data`, by having something similar to:

```
    // validate and create lot auction data
    let mut interest_value = 0; // expressed in the oracle's decimals
+   let mut seen_assets: soroban_sdk::Map<Address, bool> = map![e];
    for lot_asset in lot {
+       if seen_assets.contains_key(lot_asset.clone()) {
+           panic_with_error!(e, PoolError::InvalidLot);
+       }
+       seen_assets.set(lot_asset.clone(), true);
        // don't store updated reserve data back to ledger. This will
occur on the the auction's fill.
        // `load_reserve` will panic if the reserve does not exist
        let reserve = pool.load_reserve(e, &lot_asset, false);
        if reserve.data.backstop_credit > 0 {
            let asset_to_base = pool.load_price(e, &reserve.asset);
            interest_value += i128(asset_to_base).fixed_mul_floor(
                e,
                &reserve.data.backstop_credit,
                &reserve.scalar,
            );
```

```
                auction_data
                    .lot
                    .set(reserve.asset, reserve.data.backstop_credit);
            }
        }
```

[mootz12 (Script3) confirmed and commented]:

Validated this is a finding.

In my opinion, the impact is low for bad debt, and low/medium for interest auction. The auction system is price resistant, but not completely safe from this exploit. When an auction is sufficiently imbalanced, the step size of 0.5% is too large to allow fillers to get the "best" price. However, the exploit path is limited by the `max_positions` variable (recommended max of 10). Let's assume someone sets up a pool with 15 max positions.

In conjunction it could qualify for a medium, given there is SOME risk of bad pricing / loss, though unlikely.

### Bad Debt Auction

This is a low because the auction creator can inflate the "lot" of the auction (tokens a filler receives), but it is much less likely to occur. With 15 max positions, the worst an auction could be setup as is with 15 duplicate bid assets:

-> bid = 1 BID
-> lot = 15 BID

Thus, the breakeven fill block would be block 14:
-> bid = `1 BID * 100%` = 1 BID
-> lot = `15 BID * 7%` = 1.05 BID

Thus, the filler has adequate time to fill an auction. There is some risk that that the step size of being too large, where each block the price gets worse for the backstop gets 7.5% worse (`15x * 0.5%` step).

### Interest Auction

This is a low because the auction creator can only inflate the "bid" of the auction (tokens a filler pays). With 15 max positions, the worst an auction could be setup as is for is with 15 duplicate lot assets:

-> bid = 15 LOT
-> lot = 1 LOT

Thus, the breakeven fill block would be block 387:
-> bid = `15 LOT * (100% - 93.5%)` = 0.975 LOT

-> lot = `1 LOT * 100%` = 1 LOT

Thus, the filler has adequate time to fill the auction. The step size risk still exists here, but this has a slightly worse issue, where if the auction does not get filled before block 400 (only 13 blocks away), the lot is given away for free.
LSDan (judge) commented:

Medium is reasonable, in my opinion.

Blend mitigated:

Commit fc6a2af to block duplicate auction assets.

**Status:** Mitigation confirmed. Full details in reports from 0x007, 0xAlix2, Testerbot and oakcobalt.

---

## [M-05] Missing `update_rz_emis_data` calls in `draw` and `donate` functions lead to incorrect emissions distribution

*Submitted by 0x007, also found by 0xAlix2, adamIdarrha, carrotsmuggler, Kirkeelee, oakcobalt, rapid, Testerbot and Tigerfrake*

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/backstop/src/emissions/manager.rs#L223

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/backstop/src/backstop/fund_management.rs#L10-L42

### Finding description

The `non_queued_tokens` value is a critical component in the calculation of reward zone emissions index. While most functions that modify this value (such as deposits and withdrawals) correctly call `update_rz_emis_data` before making changes, two important functions - `execute_draw` and `execute_donate` in the `fund_management.rs` - fail to update the emissions data before modifying `non_queued_tokens`.

This oversight creates an inconsistency in the protocol's emissions accounting system, leading to unfair distribution of rewards.

### Impact

This vulnerability affects the fair distribution of emissions across pools:

- `draw`: Pools that have tokens drawn from them will unfairly earn **fewer emissions** than they deserve. This occurs because the emissions calculation would use the new (lower) token amount rather than the amount that was actually present during the `distribute` period.

- `donate`: Pools that receive donations will unfairly earn **more emissions** than they should. This occurs because the emissions index update would use the new (higher) token amount rather than the amount at the time of the last `distribute`. This would also cause the total tokens of all pools to be less than balance.

### Proof of Concept

Consider this scenario with two pools:

**Initial state:**

- PoolA: 100 tokens, 100 shares, 0 q4w
- PoolB: 100 tokens, 100 shares, 0 q4w
- Total `non_queued_tokens`: 200
- New emissions to distribute: 10
- Emissions rate increase: 0.05 per token (10/200)

**Normal case:**

1. Both pools should receive 5 emissions each (`0.05 * 100`)

**Exploited case:**

1. PoolB receives a donation of 50 tokens (increasing its tokens to 150)
2. Since `donate` doesn't call `update_rz_emis_data`, the emissions calculation uses 150 tokens instead of 100
3. When `gulp_emissions` is called, PoolB receives 7.5 emissions (0.05 * 150) instead of the fair 5 emissions
4. This results in 2.5 emissions being unfairly distributed to PoolB

This test is modified from `test_gulp_emissions`

▶ Details

### Recommended mitigation steps

Add calls to `update_rz_emis_data` at the beginning of both the `execute_draw` and `execute_donate` functions to ensure emissions calculations use the correct token amounts.

**markus_pl10 (Script3) confirmed**

**mootz12 (Script3) commented:**

Validated as a finding.

The impact of this is small, and is unlikely to impact actual claim ability in the long run. Backstop emissions data for pools is updated daily, if not more than daily, for all active Blend pools, and user's don't really have control over when `donate` and `draw` are called, so attempting to time long emission gaps up to exploit this is not possible.

However, this does result in an increased emissions output for the entire pool backstop, which does result in slightly higher emissions for users that expected. So the angle is there that if timing is lucky for a long period of time and lots of tokens are added through `donate`, we could run into an issue where some users are unable to claim.

Given this is possible and no user funds are at risk, a medium seems appropriate.
LSDan (judge) commented:

Agree that medium is more appropriate given the scale of funds lost.

mootz12 (Script3) commented:

Fixed; backstop emissions math was reverted to how it works in v1, alongside some optimizations.

- Thus, no `rz_emis_index` is tracked, and rather emissions are distributed during `distribute` based on the current state of the backstops.
- Pools can still call `distribute` immediately after a `donate`, but given pool token balances >> donate amounts, this effect is small and therefore accepted, as it does not result in any way to inflate overall emissions.

Blend mitigated:

Commit 77373e3 to remove `rz index` from backstop emissions.

**Status:** Mitigation confirmed. Full details in reports from 0x007, Testerbot, 0xAlix2 and oakcobalt.

---

## [M-06] Pools outside of the reward zone can keep receiving Blend tokens

*Submitted by Testerbot, also found by 0xabhay, 0xAlix2, adamIdarrha, aldarion, carrotsmuggler, jasonxiale, Kirkeelee, rapid, rscodes, and Tigerfrake*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/emissions/manager.rs#L228

### Finding description and impact

The Blend Protocol utilizes the `Backstop` contract as a curator system for lending pools created in the factory. The concept is that depositors in the `Backstop` contract act as a first line of capital loss if a pool accrues bad debt during its operation. These backstop depositors are incentivized through a portion of the interest charged to borrowers for each specific pool. Additionally, the system emits Blend tokens to pools that are within the reward zone.

A critical invariant in the `Backstop` system is that Blend tokens should only be emitted to pools within the reward zone. However, this report highlights a flaw that allows a pool to

continue receiving Blend emissions even after it has been removed from the reward zone, thereby completely undermining the `Backstop` system.

To understand this issue, we first examine the code responsible for removing a pool, specifically the `remove_pool()` function. Within this function, the following line is crucial:

```
// ....
set_rz_emissions(e, &to_remove, i128::MAX,to_remove_emis_data.accrued,
false);
// ....
```

This line sets the reward zone index of the removed pool to `i128::MAX`, which is intended to prevent the pool from accumulating further emissions.

Next, we consider the `update_rz_emis_data()` function, where the root cause of the issue lies:

```
// ....
if  emission_data.index <  gulp_index  ||  to_gulp {
        if  pool_balance.non_queued_tokens() > 0 {
                let  new_emissions  =
pool_balance.non_queued_tokens().fixed_mul_floor(gulp_index -
emission_data.index, SCALAR_14).unwrap_optimized();
                accrued  +=  new_emissions;
                return  set_rz_emissions(e, pool, gulp_index, accrued,
to_gulp);
} else {
        return  set_rz_emissions(e, pool, gulp_index, accrued, to_gulp);
}

// ....
}
```

In the `if` clause, the function updates the reward zone emissions information for a pool if either of the following conditions is met:

1. `emission_data.index < gulp_index`: This condition will be false since the `emission_data.index` of the pool was set to `i128::MAX` during `remove_pool()`.
2. `to_gulp == true`: This condition will be true if the current function is invoked via the `gulp_emissions()` function.

If the pool has `non_queued_tones() == 0`, then at the end of the `update_rz_emis_data` function, the code resets the index value for the pool, effectively allowing it to continue accruing Blend rewards even though it is no longer in the reward zone.

This issue is significant as it breaks the intended functionality of the `Backstop` system, allowing pools outside the reward zone to receive emissions; which will lead to the completely malfunction of the Blend protocol as there will be more Blend emitted to pools than actually Blend sent by the emitter contract as well as breaking the curation system.

### Proof of Concept

As this issue is hard to follow, I would like to present a coded proof of concept that shows its validity. Follow the next instructions to run the coded PoC:

First, we need a couple of helper functions in the main contract (these are just getter functions).

1. Add the following to the `backstop/src/contract.rs` file:

To the `trait Backstop`:

```
/////////////////////////// ADDED FOR THE PoC ////////////////////////////

fn get_user_balance(e: Env, pool: Address, user: Address) ->
UserBalance;
fn get_pool_balance(e: Env, pool: Address) -> PoolBalance;
fn get_rz_emission_data(e: Env, pool: Address) ->
storage::RzEmissionData;
fn get_rz_emission_index(e: Env) -> i128;
fn get_backstop_emission_data(e: Env, pool: Address) ->
storage::BackstopEmissionData;
fn get_user_emission_data(e: Env, pool: Address, user: Address) ->
storage::UserEmissionData;
fn get_reward_zone(e: Env) -> Vec<Address>;
fn get_last_distribution_time(e: Env) -> u64;
```

To the `impl BackstopContract`:

```
/////////////////////////// ADDED FOR THE PoC ////////////////////////////

fn get_user_balance(e: Env, pool: Address, user: Address) ->
UserBalance {
        storage::get_user_balance(&e, &pool, &user)
}

fn get_pool_balance(e: Env, pool: Address) -> PoolBalance {
        storage::get_pool_balance(&e, &pool)
}
```

```
fn get_rz_emission_index(e: Env) -> i128 {
        storage::get_rz_emission_index(&e)
}

fn get_rz_emission_data(e: Env, pool: Address) ->
storage::RzEmissionData {
        storage::get_rz_emis_data(&e, &pool).unwrap()
}

fn get_backstop_emission_data(e: Env, pool: Address) ->
storage::BackstopEmissionData {
        storage::get_backstop_emis_data(&e, &pool).unwrap()
}

fn get_user_emission_data(e: Env, pool: Address, user: Address) ->
storage::UserEmissionData {
        storage::get_user_emis_data(&e, &pool, &user).unwrap()
}

fn get_reward_zone(e: Env) -> Vec<Address> {
        storage::get_reward_zone(&e)
}

fn get_last_distribution_time(e: Env) -> u64 {
        storage::get_last_distribution_time(&e)
}
```

2. Create a file `test_blend_c4_audit.rs` into the `test-suites/tests/` folder and paste the following:

▶ Details

3. Run `make clean` and then `make build`.
4. Run `cargo test --test test_blend_c4_audit -- --nocapture --test_backstop_emissions_without_being_in_reward_zone`.

### Recommended mitigation steps

To address this issue is important to consider in the `update_rz_emis_data` function to not update the index of pools that no longer are in the reward zone.

**markus_pl10 (Script3) confirmed**

**mootz12 (Script3) commented:**

Validated this is an issue.

Note that this only affects emissions distributed by the protocol. It could cause emissions to become "unclaimable" for users, as the malicious pools would effectively capture some of the BLND going to the legit pools.

No user deposited funds are at risk.
LSDan (judge) commented:

I have a hard time seeing this as more than a medium. I would need to see maximum economic impact to bring it to high per the docs. None of the reports highlight a large impact.

mootz12 (Script3) commented:

Fixed to revert `rz_index` change to how it worked in v1 (plus some optimizations to support more reward zone pools), removing this issue.

Blend mitigated:

Commit 77373e3 to remove `rz index` from backstop emissions.

**Status:** Mitigation confirmed. Full details in reports from 0xAlix2, 0x007, Testerbot and oakcobalt.

---

## [M-07] Pool's gulped emissions could be lost if a reserve has no supply

*Submitted by 0xAlix2, also found by aldarion and alexxander*

When a pool is added to the reward zone, some emissions get distributed to it, these emissions could be "claimed" by calling `gulp_emissions` on that pool, which calls `gulp_emissions` on the backstop module. The gulped emissions are distributed to different reserves (borrow or debt reserves), according to the reserves set by the pool admin in `pool_emissions`, by calling `set_emissions_config`.

On the other hand, after gulping these emissions, users who deposited into these reserves can claim their part by calling `claim` -> `claim_emissions`, which ultimately calls `update_emission_data` and `update_user_emissions`:

```
    pub(super) fn update_emission_data(
        e: &Env,
        res_token_id: u32,
        supply: i128,
        supply_scalar: i128,
    ) -> Option<ReserveEmissionData> {
        match storage::get_res_emis_data(e, &res_token_id) {
            Some(mut res_emission_data) => {
                if res_emission_data.last_time >=
res_emission_data.expiration
```

```
                    || e.ledger().timestamp() ==
res_emission_data.last_time
                    || res_emission_data.eps == 0
@>                  || supply == 0
                {
                    return Some(res_emission_data);
                }

                // ... snip ...
            }
            None => return None, // no emission exist, no update is
required
        }
    }

    fn update_user_emissions(
        e: &Env,
        res_emis_data: &ReserveEmissionData,
        res_token_id: u32,
        supply_scalar: i128,
        user: &Address,
        balance: i128,
        claim: bool,
    ) -> i128 {
        if let Some(user_data) = storage::get_user_emissions(e, user,
&res_token_id) {
            if user_data.index != res_emis_data.index || claim {
                let mut accrual = user_data.accrued;
@>              if balance != 0 {
                    // ... snip ...
                }
                return set_user_emissions(e, user, res_token_id,
res_emis_data.index, accrual, claim);
            }
            0
        } else if balance == 0 {
            // first time the user registered an action with the asset
since emissions were added
            return set_user_emissions(e, user, res_token_id,
res_emis_data.index, 0, claim);
        } else {
            // user had tokens before emissions began, they are due any
historical emissions
            let to_accrue =
                balance.fixed_mul_floor(e, &res_emis_data.index, &
(supply_scalar * SCALAR_7));
```

```
            return set_user_emissions(e, user, res_token_id,
res_emis_data.index, to_accrue, claim);
        }
    }
```

As shown, both check if the balance/supply is `>0` to go ahead with the claiming, in other words, if no balance/supply nothing happens.

However, this check is not available when gulping and distributing the emissions between reserves, in `do_gulp_emissions`.

As a result, if a reserve has 0 supply, but it is already added in the pool emissions (and maybe not removed yet), any emissions incoming, that reserve's part will be lost forever, as it can never be claimed.

### Proof of Concept

Add the following test in `blend-contracts-v2/pool/src/emissions/manager.rs`:

```
#[test]
fn test_lost_emissions() {
    let e = Env::default();
    e.mock_all_auths();
    e.ledger().set(LedgerInfo {
        timestamp: 1500000000,
        protocol_version: 22,
        sequence_number: 20100,
        network_id: Default::default(),
        base_reserve: 10,
        min_temp_entry_ttl: 10,
        min_persistent_entry_ttl: 10,
        max_entry_ttl: 3110400,
    });

    let pool = testutils::create_pool(&e);
    let bombadil = Address::generate(&e);

    let new_emissions: i128 = 100_0000000;

    let (reserve_config, reserve_data) =
testutils::default_reserve_meta();
    let (underlying_0, _) = testutils::create_token_contract(&e,
&bombadil);
    testutils::create_reserve(&e, &pool, &underlying_0, &reserve_config,
&reserve_data);
```

```
    let (underlying_1, _) = testutils::create_token_contract(&e,
&bombadil);
    testutils::create_reserve(&e, &pool, &underlying_1, &reserve_config,
&reserve_data);

    e.as_contract(&pool, || {
        // only b_supply received emissions
        storage::set_pool_emissions(&e, &map![&e, (0, 0_5000000), (2,
0_5000000)]);

        do_gulp_emissions(&e, new_emissions);

        let mut res_emis_data_0 = storage::get_res_emis_data(&e,
&0).unwrap_optimized();
        let mut res_emis_data_2 = storage::get_res_emis_data(&e,
&2).unwrap_optimized();

        // R0 and R2 have 0 emission indexes
        assert!(res_emis_data_0.index == res_emis_data_2.index &&
res_emis_data_0.index == 0);

        // R0 and R2 have the same last_time as now
        assert!(
            res_emis_data_0.last_time == res_emis_data_2.last_time
                && res_emis_data_0.last_time == 1500000000
        );

        // R0 and R2 have the same eps
        assert!(
            res_emis_data_0.eps == res_emis_data_2.eps &&
res_emis_data_0.eps == 8267195767
        );

        e.ledger()
            .set_timestamp(e.ledger().timestamp() + (24 * 60 * 60));

        // R0 has no more d_supply
        let mut reserve_data_0 = storage::get_res_data(&e,
&underlying_0);
        reserve_data_0.d_supply = 0;
        storage::set_res_data(&e, &underlying_0, &reserve_data_0);

        // New emissions are gulped
        do_gulp_emissions(&e, new_emissions);
```

```
        res_emis_data_0 = storage::get_res_emis_data(&e,
&0).unwrap_optimized();
        res_emis_data_2 = storage::get_res_emis_data(&e,
&2).unwrap_optimized();

        // R0 index remains the same, 0
        assert_eq!(res_emis_data_0.index, 0);
        // R2 index is updated
        assert_eq!(res_emis_data_2.index, 9523809523584);

        // Both R0 and R2 last times are updated
        assert!(
            res_emis_data_0.last_time == res_emis_data_2.last_time
                && res_emis_data_0.last_time == 1500086400
        );

        // Both R0 and R2 eps are updated
        //  -> R0 should have its eps updated as it has some d_supply
        //  -> R2 shouldn't have its eps updated as it has no d_supply
(all emissions for R2 are lost, as they can't be claimed)
        assert!(
            res_emis_data_0.eps == res_emis_data_2.eps &&
res_emis_data_0.eps == 15353363558
        );
    });
}
```

### Recommended mitigation steps

If a reserve doesn't have a supply, skip its distribution, and if no reserves have a supply, block gulping emissions to avoid edge cases.

```
    fn do_gulp_emissions(e: &Env, new_emissions: i128) {
        // ... snip..

        let mut total_share: i128 = 0;
        for (res_token_id, res_eps_share) in pool_emissions.iter() {
            let reserve_index = res_token_id / 2;
            let res_asset_address =
reserve_list.get_unchecked(reserve_index);
            let res_config = storage::get_res_config(e,
&res_asset_address);

+           let reserve_data = storage::get_res_data(e,
&res_asset_address);
```

```
+           let supply = match res_token_id % 2 {
+               0 => reserve_data.d_supply,
+               1 => reserve_data.b_supply,
+               _ => panic_with_error!(e, PoolError::BadRequest),
+           };

-           if res_config.enabled {
+           if res_config.enabled && supply > 0 {
                pool_emis_enabled.push_back((
                    res_config,
                    res_asset_address,
                    res_token_id,
                    res_eps_share,
                ));
                total_share += i128(res_eps_share);
            }
        }
+       if pool_emis_enabled.len() == 0 {
+           panic_with_error!(e, PoolError::BadRequest);
+       }

        // ... snip..
    }
```

**markus_pl10 (Script3) disputed**

[mootz12 (Script3) commented]:

This is not an issue, and the suggested fix would cause more missed emissions than the current situation, in my opinion.

When a reserve has 0 supply, it's emissions data does not get updated. Once someone adds supply, they will receive all the emissions during the time it had zero supply.

If the reserve has zero supply for long enough where we need to re-write the EPS / expiration, there is nothing we can do at that point to prevent lost emissions, from the time the reserve was set to 0 supply up to the pool `gulp_emissions` time period.

Either:

1. The previous emissions are lost where 0 supply existed
2. The reserve does not receive new emissions

Given there is incentive to keep 1 as short as possible, it makes the most sense to keep the implementation as is, rather than have the reserve miss all new incoming emissions.

[LSDan (judge) decreased severity to Low and commented]:

This seems highly unlikely, albeit possible. Downgrading to low.

a_kalout (warden) commented:

The sponsor presented a couple of points that I believe aren't accurate, and I'd like to refute them.

This is not an issue, and the suggested fix would cause more missed emissions than the current situation, in my opinion.

I'm afraid that's not accurate. If the fix is implemented, any rewards that were for the 0 supply reserve would go to the other >0 reserves. Why? Because `total_share` won't account for the 0 supply, resulting in higher `new_reserve_emissions`, for all non-zero reserves:

```
let new_reserve_emissions = i128(res_eps_share)
    .fixed_div_floor(e, &total_share, &SCALAR_7)
    .fixed_mul_floor(e, &new_emissions, &SCALAR_7);
```

When a reserve has 0 supply, it's emissions data does not get updated. Once someone adds supply, they will receive all the emissions during the time it had zero supply.

This is also not accurate. When emissions come in for 0 supply reserves, the emission data is indeed not updated (emission data index is not updated); however, the EPS is updated, and this is shown in the above PoC. When someone adds some supply to a reserve that has "lost" emissions, his emission index would be set to the current supply's emission index, in `update_user_emissions`. Later, when he tries to claim the "lost" emissions, it'll be calculated as follows:

```
if user_data.index != res_emis_data.index || claim {
    let mut accrual = user_data.accrued;
    if balance != 0 {
        let delta_index = res_emis_data.index - user_data.index;
        require_nonnegative(e, &delta_index);
        let to_accrue = balance.fixed_mul_floor(
            e,
            &(res_emis_data.index - user_data.index),
            &(supply_scalar * SCALAR_7),
        );
        accrual += to_accrue;
    }
    return set_user_emissions(e, user, res_token_id, res_emis_data.index,
accrual, claim);
}
```

Will `user_data.index` be != `res_emis_data.index`? No. So, no, all "lost" emissions are lost forever and can't be claimed. Well.. if new emissions come in with supply > 0 (after the

deposit), only the new ones are claimable.

If the reserve has zero supply for long enough where we need to re-write the EPS / expiration, there is nothing we can do at that point to prevent lost emissions, from the time the reserve was set to 0 supply up to the pool `gulp_emissions` time period.

I am not sure about the added value of re-writing the EPS/expiration. The recommended mitigation handles this perfectly, even if the reserve doesn't receive emissions for ages, without any losses.

Regarding the likelihood, first, I'd like to point out that for a specific asset, we have 2 reserves, b (supply) and d (debt). Having a 0 b supply is very unlikely, I agree; however, having a 0 d supply isn't that rare. It's pretty normal in lending protocols to have no loans issued for a specific asset.

As a result, I respectfully believe this easily guarantees at least a medium severity. I would appreciate it if the judge could take another look at this.
LSDan (judge) commented:

Thank you for the clarifications. On balance, I agree that the 0 d supply may not be super rare, but this still requires the admin to configure the protocol to send emissions to a pool with no activity, does it not?

a_kalout (warden) commented:

Hmm, not really. The emissions could be set to different reserves, for example, reserve X, which corresponds to a d supply (d or b depends on the key, whether even or odd), and the supply would go down to 0 (repay, withdraw, ...), all emissions after that are lost, until the admin removes that from the emissions array.

LSDan (judge) increased severity to Medium and commented:

Ok. I see the point. Reconstituting this as a medium.

---

## [M-08] Removing a pool from the reward zone leads to the loss of ungulped emissions

*Submitted by 0xAlix2, also found by 0xadrii, carrotsmuggler, oakcobalt, and Testerbot*

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/backstop/src/emissions/manager.rs#L66-L78

### Finding description and impact

Pools enter the reward zone in the backstop module to search emissions. When distribution happens, the global reward zone index is updated. Later, when changes happen to different pools, the pool reward zone index is updated to account for the newly distributed emissions. Each pool takes its part and adds it to the `accrued` balance until gulped later.

On the other hand, pools could be removed from the reward zone, if the pool's backstop balance falls below a certain threshold. When a pool is removed, `remove_from_reward_zone` is called, which also calls `remove_pool`, which in turn removes the pool from the reward zone array, and sets the pool's reward zone index to infinity.

https://github.com/code-423n4/2025-02-blend/blob/main/blend-contracts-v2/backstop/src/emissions/manager.rs#L92-L95

```
let to_remove_emis_data = storage::get_rz_emis_data(e,
&to_remove).unwrap_optimized();
set_rz_emissions(e, &to_remove, i128::MAX, to_remove_emis_data.accrued,
false);

reward_zone.remove(idx);
```

However, this doesn't account for the unaccrued pool's emissions, forcing them to be lost, in other words, `update_rz_emis_data` is not called to account for those emissions.

For example, if a distribution is made, and without any pool balance changes happen, the pool is removed. The pool's part of the last emission is lost, and can never be claimed.

### Proof of Concept

Add the following test to `blend-contracts-v2/backstop/src/emissions/manager.rs`:

▶ Details

### Recommended mitigation steps

Accrue the emissions of the removed pool, before setting its rewards index to infinity.

```
    /// remove a pool to the reward zone if below the minimum backstop
deposit threshold
    pub fn remove_from_reward_zone(e: &Env, to_remove: Address) {
        let mut reward_zone = storage::get_reward_zone(e);

        // ensure to_add has met the minimum backstop deposit threshold
        // NOTE: "to_add" can only carry a pool balance if it is a
deployed pool from the factory
        let pool_data = load_pool_backstop_data(e, &to_remove);
        if require_pool_above_threshold(&pool_data) {
            panic_with_error!(e, BackstopError::BadRequest);
        } else {
+           update_rz_emis_data(e, &to_remove, false);
            remove_pool(e, &mut reward_zone, &to_remove);
```

```
                storage::set_reward_zone(e, &reward_zone);
            }
        }
```

markus_pl10 (Script3) disputed

mootz12 (Script3) commented:

This is not a finding.

Removing pools is safeguarded by a check to ensure distribution was called fairly recently, to limit the lost emissions for removed pools.

This is not a common code pathway, and losing a day of emissions is not considered vital, due to the complications of forcing distribute to be invoked during removal with resource limitations.

However, documentation should be added to ensure that is clear.

LSDan (judge) commented:

I disagree. This locks funds and causes a loss of emissions. Medium is appropriate here, per the docs.

Blend mitigated:

Commit 6204dba to improve reward zone changes emissions impact.

**Status:** Mitigation confirmed. Full details in reports from oakcobalt, 0xAlix2, 0x007 and Testerbot.

---

## [M-09] When code defaults on remaining liability, it does not delete remaining auction, which is problematic if the user has called fill with a % less than 100

*Submitted by rscodes, also found by 0xtheauditor, aldarion, and klau5*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/auctions/bad_debt_auction.rs#L131

### Finding description

```
pub fn fill_bad_debt_auction(...) {
    ....

    // bid only contains d_token asset amounts
```

```
    backstop_state.rm_positions(e, pool, map![e],
auction_data.bid.clone());
    filler_state.add_positions(e, pool, map![e],
auction_data.bid.clone());

    ....

    // If the backstop still has liabilities and less than 5% of the
backstop threshold burn bad debt
    if !backstop_state.positions.liabilities.is_empty() {
        let pool_backstop_data =
backstop_client.pool_data(&e.current_contract_address());
        let threshold =
calc_pool_backstop_threshold(&pool_backstop_data);
        if threshold < 0_0000003 {
            // ~5% of threshold
            let reserve_list = storage::get_res_list(e);
            for (reserve_index, liability_balance) in
backstop_state.positions.liabilities.iter() {
                let res_asset_address =
reserve_list.get_unchecked(reserve_index);
                let mut reserve = pool.load_reserve(e,
&res_asset_address, true);
                backstop_state.default_liabilities(e, &mut reserve,
liability_balance);
                pool.cache_reserve(reserve);

                PoolEvents::defaulted_debt(e, res_asset_address,
liability_balance);
            }
        }
    }
    backstop_state.store(e);
}
```

We can see that in the bottom of the code, if the backstop threshold goes below the certain `%` then it deletes the remaining liability by calling `backstop_state.default_liabilities`.

However if we look at the function `fill` in `auction.rs` where users can pass in a variable `percent_filled`.

```
pub fn fill(
    ...
--> percent_filled: u64,
) -> AuctionData {
```

```
    ...
    let auction_data = storage::get_auction(e, &auction_type, user);
--> let (to_fill_auction, remaining_auction) = scale_auction(e,
&auction_data, percent_filled);
    match AuctionType::from_u32(e, auction_type) {
        AuctionType::UserLiquidation => {
            fill_user_liq_auction(e, pool, &to_fill_auction, user,
filler_state)
        }
        AuctionType::BadDebtAuction => {
            fill_bad_debt_auction(e, pool, &to_fill_auction,
filler_state)
        }
        AuctionType::InterestAuction => {
            fill_interest_auction(e, pool, &to_fill_auction,
&filler_state.address)
        }
    };

    if let Some(auction_to_store) = remaining_auction {
        storage::set_auction(e, &auction_type, user, &auction_to_store);
    } else {
        storage::del_auction(e, &auction_type, user);
    }

    to_fill_auction
}
```

So, if the user fills with like 50%, for example, `scale_auction` only passes along like 50% of the auction data to `fill_bad_debt_auction`, the remaining 50% is stored in `remaining_auction` and stored back into `storage::set_auction`.

Hence, right now the bug is that when the threshold goes below 5%, the code deletes the liability by defaulting on it, but does not delete any remaining auction, which allows users to steal backstop tokens.

### Sequence

Let's explore what happens during such a case.

Bid token - the `dtokens` (debt) the auction winner will take on

- From block 0 to 200, remains the same at the peak value. Then from block 200 to 400 begins to decrease. After block 400, its just zero

Lot token - The backstop token the pool will transfer to the auction winner.

- From block 0 to 200, the amount of lot tokens given to winner scales upwards and peaks at block 200. Then from then on stays the same at that peaked value.

1. Suppose there is a bad debt auction.
2. `Alice` fills the bad debt auction, calling `fill` with `percent_filled = 50%`.
3. It goes below the threshold and the code deletes backstop remaining liability.
4. However, the code does not delete the 50% `remaining_auction` which is now set in the bad debt auction storage key (also prevents new bad debt from being created btw).
5. From now, all the way till block 400, **no auctioner can fill the auction**. This is because < block 400, the bid token is non-zero, and since the code defaulted the liability under backstop, the line `backstop_state.rm_positions` will panic.

Hence, the auction can only be filled after block 400. **But, now the issue is,** when the auction is filled at block 400 onwards, the user who fills the auction does not take on any debt in place of the backstop, but still **gets the lot amount of backstop tokens for FREE** (at the expense of backstop stakers).

### Impact

This breaks the normal flow of the bad debt auction and causes serious loss for backstop stakers. If you think about, of course every auction can be like this, where everyone waits until block 400 to get the backstop token **for free, at the loss of the backstop stakers**.

**However, that does not happen due to the free market the code facilitates.** Once the exchange becomes worth it, auctioners will rush to do the exchange, taking on debt in exchange for the backstop token because **they know that if they don't, then others will anyway**.

But in this case, the fact that the code removes liability from backstop but not the `remaining_auction` means that **no one** can fill the auction before block 400 because **it will always panic due to the** `rm_position`, which now breaks the free market flow of regular bad debt auctions.

- This loophole allows the first user to call `fill` at block 400 to **steal** all the backstop tokens in remaining lot amount **without taking on any debt**. (Which is also **despite the backstop already dropping below threshold**)

### PoC

PoC is written here.

### Recommendation

Delete `remaining_auction` if the code is going to default liabilities. So that backstop stakers don't lose tokens **for nothing** (since there isn't any liability anymore) when the threshold is below 5% already.

markus_pl10 (Script3) confirmed

mootz12 (Script3) commented:

Validated this is an issue.

The 5% threshold was chosen as a "dust" limit where the expectations of the pool was that it could get up to that amount of backstop funds to cover its defaulted debt, otherwise, it was on it's own. The pool only expects that much coverage, and the backstop assumes all it's tokens can be used to cover bad debt.

This does cause some unnecessary loss for suppliers, as generally fillers would have used the rest of the funds in the auction to cover additional bad debt.

However, the worst case loss for suppliers and/or backstop depositors does not change, it just opens a scenario where both parties can have a worst case experience, at the gain of a filler.

LSDan (judge) commented:

Good finding, but it requires a series of conditions to line up just right, so I can't see it exceeding medium.

mootz12 (Script3) commented:

Fixed to:

- Only process post auction actions like bad debt and defaulting if the auction was fully filled, and no remaining auction exists.
- Re-work `bad_debt` to handle any edge case where auction leaves an account in a state where debt can be either marked as bad debt or defaulted.

Blend mitigated:

Commit 59acbc9; Improvements to assign bad debt after user liquidations and clean up post liquidation actions.

**Status:** Mitigation confirmed. Full details in reports from 0xAlix2, Testerbot, 0x007, oakcobalt and rscodes.

---

## [M-10] Division before multiplication may cause division by zero DOS during low backstop supply

*Submitted by oakcobalt*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/auctions/backstop_interest_auction.rs#L76-L84

### Finding description and impact

Division before multiplication and unchecked oracle decimals may cause division by zero DOS during low backstop supply.

### Proof of Concept

When calculating `backstop_token_to_base`, division before multiplication is performed when deriving `backstop_value_base`. Currently, `oracle_scalar` is intended to prevent excessive precision loss in `backstop_value_base` calculation.

There are two vulnerabilities here:

1. Pool deployment is permissionless and `oracle_scalar` is based on unchecked pool's constructor parameters, i.e. custom oracle implementation `oracle_client.decimals()`. There is no check that the oracle's decimal is greater than 7. When `oracle_scalar` is less than `10e7`, `backstop_value_base` will round down and lose precision.

```
pub fn execute_initialize(
    e: &Env,
    admin: &Address,
    name: &String,
    oracle: &Address,
    bstop_rate: &u32,
    max_positions: &u32,
    min_collateral: &i128,
    backstop_address: &Address,
    blnd_id: &Address,
) {
    let pool_config = PoolConfig {
|>      oracle: oracle.clone(), //@audit no check on oracle decimal
precisions.
        min_collateral: *min_collateral,
        bstop_rate: *bstop_rate,
        status: 6,
        max_positions: *max_positions,
    };
```

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/config.rs#L28

2. Division before multiplication: `backstop_value_base` performs integer division first before multiplication in `backstop_token_to_base`.

```
    // get value of backstop_token (BLND-USDC LP token) to base
    let pool_backstop_data =
backstop_client.pool_data(&e.current_contract_address());
    //@audit (usdc * oracle_scalar / scalar_7) perform division first
before multiplication in `backstop_token_to_base`
|>  let backstop_value_base = pool_backstop_data
        .usdc
        .fixed_mul_floor(e, &oracle_scalar, &SCALAR_7) // adjust for
oracle scalar
        * 5; // Since the backstop LP token is an 80/20 split of
USDC/BLND, we multiply by 5 to get the value of the BLND portion
    let backstop_token_to_base =
        backstop_value_base.fixed_div_floor(e,
&pool_backstop_data.tokens, &SCALAR_7);

    // determine lot amount of backstop tokens needed to safely cover bad
debt, or post
    // all backstop tokens if there isn't enough to cover the bad debt
    let mut lot_amount = debt_value
        .fixed_mul_floor(e, &1_2000000, &SCALAR_7)
        .fixed_div_floor(e, &backstop_token_to_base, &SCALAR_7);
```

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/auctions/bad_debt_auction.rs#L75-L81

The above may cause `backstop_token_to_base` round down to zero when backstop is low in supply, resulting in division by zero panic in `create_interest_auction_data` and `create_bad_debt_auction_data`. Backstop supply(`pool_backstop_data.tokens`) can be low when lending pool is new or due to large backstop depositor withdrawals.

### POC

Suppose `pool_backstop_data.usdc = 0_0050000`, `pool_backstop_data.tokens = 10_0000000`, `oracle_scalar = 100`.

`backstop_value_base = (pool_backstop_data.usdc * oracle_scalar / SCALAR_7) * 5`

```
= (50000 * 100 / 1e7) * 5 -> 0
```

`backstop_token_to_base = backstop_value_base * SCALAR_7 / pool_backstop_data.tokens`

```
= 0
```

**Case 1: bad debt auction:**
`lot_amount = debt_value * 1.2 * SCALAR_7 / backstop_token_to_base` -> division by zero

**Case 2: interest auction:**
`bit_amount = interest_value * 1.2 * SCALAR_7 / backstop_token_to_base` -> division by zero

As seen above, both bad debt auction and interest auction flow can be DOSsed when backstop supply is critically low.

### Impact

When backstop token supply for a pool is critically low (or zero), the pool can be extremely unhealthy and risky. The pool needs more backstop deposits through interest auctions where existing backstop credits accumulated in the pool's reserves can be auctioned in exchange for more backstop tokens to offset bad debts.

However, from the POC we see that interest auction can be DOSsed and the pool cannot attract more backstop tokens when it needs them the most.

### Recommended mitigation steps

In pool's constructor -> `require_valid_pool_config`, consider enforcing checks that the oracle decimal is no less than 7.

May also consider fetching the comet pool's usdc and total supply to derive `backstop_token_to_base` directly without relying on the intermediate `pool_backstop_data.tokens`.

**markus_pl10 (Script3) confirmed**

**mootz12 (Script3) commented:**

Validated this is an issue. Documentation likely needs to be added to ensure extremely low decimal oracles aren't used.

Regardless, an interest auction likely should be able to be created for an empty backstop, even if this situation is unlikely (can't borrow funds unless your past backstop threshold).

No risk to backstop credit, as if more tokens enter the backstop, the credit is safely tracked and the interest auction can be created later. This is incredibly likely as interest auctions need at least `$200` to get kicked off, and this example shows `$0.05` in total backstop deposits.

**LSDan (judge) commented:**

This holds as a medium for me. The functionality of the protocol could be impacted enough that mitigations are required.

mootz12 (Script3) commented:

Fixed here - token price returned from backstop, no longer depends on pool backstop state.

Testerbot (warden) commented:

I would like to address the classification of this issue, as I believe it may have been incorrectly assessed.

The issue in question was marked as medium severity. However, I would like to provide my reasoning for reconsideration:

The core of the reported issue lies in the following formula:

```
backstop_value_base = (pool_backstop_data.usdc * oracle_scalar /
SCALAR_7)
```

If the product of `pool_backstop_data.usdc * oracle_scalar` is less than `10e7`, then `backstop_value_base` will round down to zero, leading to a division by zero panic later in the code.

### Likelihood Analysis

The condition to trigger this issue is `pool_backstop_data.usdc * oracle_scalar < 10e7`.

In Soroban, tokens typically have 7 decimal places, which applies to USDC. Therefore, the first condition to meet is for `pool_backstop_data.usdc` to be less than `1 USDC` (10e7). This means that at least 1 USDC will prevent this issue from occurring.

If we consider `0.1 USDC` (10e6), the oracle price would need to have at most 1 decimal (10). For `0.01 USDC` (10e5), the oracle price would need at most 2 decimals (100), and so on.

Consequently, it is unlikely that such a small amount of USDC would be deposited in the backstop, and having oracle prices with such limited decimal places simultaneously is even less probable. From the currently deployed oracle providers on Stellar, I have not found any prices with fewer than 7 decimals:

Current oracle providers.
Reflector contract. Reflector set prices with 7 decimals.

### Impact Analysis

Even in the unlikely event that this situation occurs, the impact does not warrant a medium severity classification. The report describes the following impact:

The above may cause `backstop_token_to_base` round down to zero when backstop is low in supply, resulting in division by zero panic in `create_interest_auction_data` and `create_bad_debt_auction_data`.

The division by zero will prevent the creation of bad debt and interest auctions. Some facts about this:

- These functions are not core components of the lending protocol that put user funds at risk.
- The denial of service (DoS) regarding the interest auction is temporary and easily remedied; an admin or any user can simply make a deposit to the backstop contract, which is not affected by this issue.
- This situation is no different to a pool without backstop deposits, where it is normal not to have bad debt or interest auctions (as there would be no one to cover bad debt or claim interest).

In conclusion, we are dealing with a scenario that is very unlikely to occur, and the worst-case impact is merely a temporary DoS.
rscodes (warden) commented:

In addition to Testerbot's comment, I would like to add that according to the competition page token table: `Low decimals ( < 6) | Out of scope`.

I understand that the above submission is about oracle decimal. However, `oracle decimal => token decimal`. If low decimal tokens are not used (as oos) then it doesn't make sense to use a low decimal oracle, since the oracle is supposed to reflect the price of the token.

LSDan (judge) commented:

Ruling stands. This is 6, not `< 6`. Medium still holds for me.

Blend mitigated:

Commit f0296cf to make `lp token` valuation pool independent for auction creation.

**Status:** Mitigation confirmed. Full details in reports from 0xAlix2, Testerbot, 0x007, oakcobalt and rscodes.

---

## [M-11] Fee-vault can be made insolvent in case of defaults

*Submitted by carrotsmuggler, also found by 0x007, Tigerfrake, and YouCrossTheLineAlfie*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/fee-vault/src/contract.rs#L310-L311

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/fee-

vault/src/reserve_vault.rs#L72-L74

### Finding description and impact

In case of a loan default, the backstop funds are insufficient to cover the liability. In this case, the hit is taken by the collateral providers, and the `b_rate` drops.

```
pub fn default_liabilities(&mut self, e: &Env, reserve: &mut Reserve,
amount: i128) {
    self.remove_liabilities(e, reserve, amount);
    // determine amount of funds in underlying that have defaulted
    // and deduct them from the b_rate
    let default_amount = reserve.to_asset_from_d_token(e, amount);
    let b_rate_loss = default_amount.fixed_div_ceil(&e,
&reserve.data.b_supply, &SCALAR_12);
    reserve.data.b_rate -= b_rate_loss;
```

In this scenario, the `b_rate` of the token is reduced. This creates a situation in the fee vault which can lead to insolvency.

The fee-vault, during deposits and withdrawals, calls `update_rate` to make sure it has the latest `b_rate` from the pool.

```
let now = e.ledger().timestamp();
if now == self.last_update_timestamp {
    return;
}

let new_rate = pool::reserve_b_rate(e, &self.address);
if new_rate == self.b_rate {
    self.last_update_timestamp = now;
    return;
}
```

However, as can be seen above, there is an escape condition where the `b_rate` is not updated if the timestamp is the same. This can trigger when there are two transactions that are executed in the same block. In that case, the `b_rate` is not updated and the older value is used.

This is not an issue in general operations, because `b_rate` cannot change within the same block. However, in the case of a default, the `b_rate` can change within the same block. So the `fee_vault` can be made to use an outdated `b_rate`.

Imagine a situation where 3 transactions are bundled together in the same block:

1. tx 1 - a simple deposit to the fee vault
2. tx 2 - a liquidation causing a default
3. tx 3 - a withdrawal from the fee vault

Tx 1 updates the `b_rate` of the vault to the one in the pool. Tx 2 drops the `b_rate` in the pool, but the one in the vault remains the same. Tx 3 uses the outdated higher `b_rate` to calculate withdrawal amounts, since tx1 was in the same block so `now == self.last_update_timestamp` check passes. Since a higher `b_rate` is used, the fee vault ends up burning more b_tokens than it expects, leading to insolvency.

Lets assume the `b_rate` drops from 1.1 to 1.05. In tx 3, the vault still uses 1.1, but the pool uses `1.05`. Say a user wants to withdraw 1000 amount from the vault. The `pool::withdraw(&e, &reserve, &user, amount);` call will burn `1000/1.05=953` `b_tokens` from the vault. But in the vaults `withdraw` function, the `b_tokens_amount` will be calculated as `1000/1.1=910`. So the `vault.total_b_tokens` will be reduced by 910, but the pool will burn 953 btokens from the vault. So no there is a deficit of 43 `b_tokens` in the vault, which leads to the insolvency for the user who exits last.

```
let b_tokens_amount = vault.underlying_to_b_tokens_up(amount); //@audit
older b_rate used

let mut user_shares = storage::get_reserve_vault_shares(e,
&vault.address, user);
let share_amount = vault.b_tokens_to_shares_up(b_tokens_amount);
require_positive(e, share_amount, FeeVaultError::InvalidBTokensBurnt);

if vault.total_shares < share_amount || vault.total_b_tokens <
b_tokens_amount {
    panic_with_error!(e, FeeVaultError::InsufficientReserves);
}

if share_amount > user_shares {
    panic_with_error!(e, FeeVaultError::BalanceError);
}
vault.total_shares -= share_amount;
vault.total_b_tokens -= b_tokens_amount; //@audit reducing by incorrect
amount
```

All three transactions above can be sent by the same person, bundled together, removing the MEV necessity.

### Proof of Concept

Say the default reduces the `b_rate` from 1.1 to 1.05. During withdrawal, the pool has a `b_rate` of 1.05, but the vault has a `b_rate` of 1.1.

Thus, when withdrawing 1000, the pool will burn `1000/1.05=953` `b_tokens`, but the vault will burn `1000/1.1=910` `b_tokens`. This leads to a deficit of 43 `b_tokens` in the vault, which leads to insolvency for the user who exits last.

### Recommended mitigation steps

Update the rate even if the timestamp matches.

**markus_pl10 (Script3) confirmed**

**mootz12 (Script3) commented:**

Validated this is a finding. The execution of this is extremely edge case, as A (the backstop needs to hit the doomsday scenario of running out of funds), and B, you have to win the bad debt auction to bundle the exploit pathway correctly in one contract call.

However, it doesn't seem like there is a way to extract funds here. An attacker would need to already be using the fee vault, and use the exploit to avoid losing funds, at the expense of the last withdrawer.

**mootz12 (Script3) commented:**

Fixed to load `b_rate` from chain every time, and update the internal vault every time.

**Blend mitigated:**

Commit a63d9a0 to patch `b_rate` loss sandwhich issue.

**Status:** Mitigation confirmed. Full details in reports from 0x007, oakcobalt, 0xAlix2 and Testerbot.

---

## [M-12] Malicious actors can repeatedly dilute emissions to a longer timeframe

*Submitted by rscodes*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/contract.rs#L502-L508

### Summary

```
pub fn gulp_emissions(e: &Env) -> i128 {
    let backstop = storage::get_backstop(e);
    let new_emissions =
```

```
        BackstopClient::new(e,
&backstop).gulp_emissions(&e.current_contract_address());
    do_gulp_emissions(e, new_emissions);
    new_emissions
}
```

In the pool folder contracts, `gulp_emissions` allow any users to call it. It will then fetch the emissions from the backstop pool, record it under the variable `new_emissions` and distribute it with `do_gulp_emissions(e, new_emissions)`.

`do_gulp_emissions` then calls `update_reserve_emission_eps` to adjust the new timeframe and new eps.

```
fn update_reserve_emission_eps(
    e: &Env,
    reserve_config: &ReserveConfig,
    asset: &Address,
    res_token_id: u32,
    new_reserve_emissions: i128,
) {
    ....
    let expiration: u64 = e.ledger().timestamp() + 7 * 24 * 60 * 60;
//@my_comment <---- new expiration date

    if let Some(mut emission_data) = distributor::update_emission_data(
        e,
        res_token_id,
        supply,
        10i128.pow(reserve_config.decimals),
    ) {
        // data exists - update it with old config

        if emission_data.last_time != e.ledger().timestamp() {
            // force the emission data to be updated to the current
timestamp
            emission_data.last_time = e.ledger().timestamp();
        }
        // determine the amount of tokens not emitted from the last
config
        if emission_data.expiration > e.ledger().timestamp() {
            let time_since_last_emission = emission_data.expiration -
e.ledger().timestamp();

            // Eps is scaled by 14 decimals
```

```
            let tokens_since_last_emission =
i128(emission_data.eps).fixed_mul_floor(
                e,
                &i128(time_since_last_emission),
                &SCALAR_7,
            );
-->         tokens_left_to_emit += tokens_since_last_emission;
        }

-->     let eps = u64(tokens_left_to_emit * SCALAR_7 / (7 * 24 * 60 *
60)).unwrap_optimized();

        emission_data.expiration = expiration;
        emission_data.eps = eps;
        storage::set_res_emis_data(e, &res_token_id, &emission_data);
        PoolEvents::reserve_emission_update(e, res_token_id, eps,
expiration);
    } else {
        ....
    }
}
```

As shown by the code, it will calculate the **new expiration time** and calculate the previous **un-distributed** rewards before diluting it over the new expiration timestamp.

There is no restriction on the minimum time that has to pass before this function can be called. If we look at `do_gulp_emissions` we can see that the only restriction is on the value of `new_emissions`. (Comment is the original comment from the code).

```
fn do_gulp_emissions(e: &Env, new_emissions: i128) {
  // ensure enough tokens are being emitted to avoid rounding issues
  if new_emissions < SCALAR_7 {
      panic_with_error!(e, PoolError::BadRequest)
  }
  ...
}
```

The emission token has 7 decimal places, so that means this can be called whenever there is **1 token to be emittted** with **no minimum time** that must pass before it could be called again.

### Impact

This bug has been reported before in some audit's and accepted as a Medium in such as the [Infrared audit on Cantina](), as well as the Loopfi July Code4rena audit.

Basically, the impact is that since there is no minimum time that must pass before the function can be called again, anyone can **repeatedly dilute the existing rewards into a longer timeframe to delay the emissions of the reward**.

Users who do not have free funds now, and only plan to deposit sometime in the near future can consistently carry out this attack (like say every 3 minutes) and cause existing rewards to get diluted into the future timeframe. (**and that's extremely unfair to current stakers**).

### Recommended mitigation steps

Add a minimum time duration that must pass before `gulp_emissions` can be called again. That way users cannot repeatedly dilute the rewards into a longer timeframe which is unfair for current stakers.

**markus_pl10 (Script3) disputed**

**mootz12 (Script3) commented:**

This is probably not an issue. There is a minimum time that must pass before distribute can be called on the backstop (the source for all emissions), which helps mitigate this, as it can only be called once per hour. Also, the min balance check on `gulp_emissions` prevents emissions from being updated after emissions have stopped being directed towards a pool.

Even in the case where a specific reserve loses it's emission status, it will not have it's emissions config updated.

The only case this might impact something is when a reserve gets added, and spamming `gulp_emissions` every hour might make it take longer for the emissions to "reach full steam". However, I'm not confident this is true.

**LSDan (judge) commented:**

This is a hand-wavy hypothetical report that does point to an issue where the protocol would not function as expected and there may be loss of unmatured / unrealized yield (reference). As such, it fits as a valid medium.

**mootz12 (Script3) commented:**

Fixed to:

- Create better bounds on when distribute and gulp are run.
- This fix was primarily for cleaning up reward zone changes, but does fix this as well, given gulp can only be run once a day per pool.

**Blend mitigated:**

Commit 6204dba to improve reward zone changes emissions impact.

**Status:** Mitigation confirmed. Full details in reports from oakcobalt, 0xAlix2, 0x007 and Testerbot.

---

## [M-13] Missing reserve interest accrual prior to backstop take rate update leads to incorrect `backstop_credit` computation

*Submitted by Oxadrii, also found by oakcobalt, Testerbot, and Tigerfrake*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/config.rs#L51

### Summary

Reserve interest is not updated when `bstop_rate` is changed. Because `bstop_rate` determines the percentage of the accrued interest that will be stored as backstop credit, not accruing the reserve prior to changing the `bstop_rate` will make previous unaccrued assets too.

### Vulnerability details

When loading a reserve and accruing interest, some of the `accrued_interest` is stored as backstop credit, which will later be auctioned via an interest auction. The amount of interest going to the backstop credit is determined by the `bstop_rate`, which is configured as a percentage for the reserve in question, and computed as `accrued.fixed_mul_floor(e, &i128(bstop_rate), &SCALAR_7);`:

```
// File: reserve.rs

pub fn load(e: &Env, pool_config: &PoolConfig, asset: &Address) ->
Reserve {
        ...

        reserve.accrue(e, pool_config.bstop_rate, accrued_interest);

        reserve.data.last_time = e.ledger().timestamp();
        reserve
    }

 fn accrue(&mut self, e: &Env, bstop_rate: u32, accrued: i128) {
        let pre_update_supply = self.total_supply(e);

        if accrued > 0 {
            let mut new_backstop_credit: i128 = 0;
            if bstop_rate > 0 {
                new_backstop_credit = accrued.fixed_mul_floor(e,
&i128(bstop_rate), &SCALAR_7);
                self.data.backstop_credit += new_backstop_credit;
```

```
            }

            self.data.b_rate = (pre_update_supply + accrued -
new_backstop_credit).fixed_div_floor(
                e,
                &self.data.b_supply,
                &SCALAR_12,
            );
        }
    }
```

`bstop_rate` can be updated for a specific reserve by calling `execute_update_pool()`:

```
// File: config.rs

pub fn execute_update_pool(
    e: &Env,
    backstop_take_rate: u32,
    max_positions: u32,
    min_collateral: i128,
) {
    let mut pool_config = storage::get_pool_config(e);
    pool_config.bstop_rate = backstop_take_rate;

    ...
}
```

The problem is that updating the `bstop_rate` does not accrue interest for the reserve. This can lead to the following situation:

1. A pool has a total of 100 tokens of a given reserve to be accrued as `accrued_interest`. The current `bstop_rate` is configured to 30%, so 30 tokens should be stored as `backstop_credit` for the corresponding period.
2. The admin of the pool triggers a pool update in order to change the `bstop_rate` from 30% to 0%. Note there's still not been any accrual in the reserve (this can be due to low interactions in the pool, for example), and the pool updating logic doesn't accrue interest for the reserve either. This effectively directly sets the `bstop_rate` to 0%.
3. After changing the `bstop_rate`, an interaction is performed in the pool, and the interest of the reserve is accrued. However, because `bstop_rate` was changed, a 0% of the 100 accrued will be directed to the `backstop_credit`, although a 30% should have been directed to it, as the 100 tokens were accrued while the backstop rate config was set to 30%.

### Impact

Medium. If `bstop_rate` changes, the amount of interest allocated to `backstop_credit` can be miscalculated. This issue is most likely to occur in pools with lower activity, where the longer intervals between interactions allow unaccounted rewards to accumulate. In contrast, pools with higher activity face a reduced risk, as interest accrues with each interaction that triggers one of the main flows.

### Recommended mitigation steps

Consider loading the reserve and storing it so that interest is accrued prior to updating the `bstop_rate`. This will make the old `bstop_rate` value be used to compute the corresponding interest.

**markus_pl10 (Script3) confirmed**

**mootz12 (Script3) commented:**

Fixed here to update reserves when backstop take rate changes.

**Blend mitigated:**

Commit b3e5af4 to improve pool config admin functions.

**Status:** Mitigation confirmed. Full details in reports from Testerbot, oakcobalt, 0xAlix2 and 0x007.

---

## [M-14] Attackers can maliciously inflate `total_supply` temporarily to exceed utilization rate limit and push the pool towards 100% util rate, potentially causing a loss of lender funds

*Submitted by rscodes, also found by 0x37*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/actions.rs#L382

### Summary

Blend pools have a utilisation rate limit that borrowers are not allowed to cross. Lenders deposit into the pool because they agree with the utilization limit set that cannot be crossed to protect them from bad debt losses.

However, a malicious user can bypass it with the following attack vector and push the pool illegally towards a 100% util rate (past the limit), causing potential loss for lenders during volatile market conditions

Currently during a borrow action, the only part where the utilization rate is checked is in `apply_borrow`, which checks it by calling the function `reserve.require_utilization_below_max(e)`.

However, there is a way where users can build transactions to simply bypass the utilization rate. Consider the following transaction order.

In the build actions, the user can build as (assuming the malicious user already has collateral to make a borrow):

1. `apply_supply`
2. `apply_borrow`
3. `apply_withdraw`

(This is done in the same build transaction).

This works because `reserve.require_utilization_below_max(e)` is `total_liabilities / total_supply`. You can see that the transaction **temporarily inflates** `total_supply` **before withdrawing it**.

**Important note:** Even though, `apply_borrow` sets `check_health` to `true`, check health **does not** check the utilization rate. It checks whether the individual user has sufficient collateral.

### Impact

This allows the agreed upon utilization rate to be bypassed, putting the pool at a higher risk at baddebt than the lenders have agreed upon when they deposited their funds.

Malicious users can do this during volatile market conditions, **to bring utilisation rate to 100%**, which is almost guaranteed to cause a loss of lender funds from baddebt in volatile market conditions. Hence, this is a high impact by Code4rena rules.

### Likelihood

There isn't much precondition to carry out this attack to bypass util rate. In fact, the attacker doesn't even need to have the funds that they use to temporarily inflate `total_supply`. This is because funds are net-off and pulled in at the end, so the withdraw will net-off away from the supply.

The attacker will be able to continue having the loan, even though the loan may be causing the pool to reach **100% utilization rate**.

### Proof Of Concept (POC)

Go to `actions.rs` in the V2 repo. (I'm using the repo provided by the C4 audit page).

There is currently a function called `test_build_actions_from_request_borrow_errors_over_max_util()` already coded by

the sponsor. It has a `#[should_panic....]` which is meant to catch the panic when the user borrows over util rate.

First, lets remove the `#[should_panic....]` line as I will now prove the malicious user can avoid the panic by doing this hack.

Change the test function to the following:

```
#[test]
fn test_build_actions_from_request_borrow_errors_over_max_util() {
    let e = Env::default();
    e.mock_all_auths();

    let bombadil = Address::generate(&e);
    let samwise = Address::generate(&e);
    let pool = testutils::create_pool(&e);

    let (underlying, _) = testutils::create_token_contract(&e,
&bombadil);
    let (mut reserve_config, mut reserve_data) =
testutils::default_reserve_meta();
    reserve_config.max_util = 0_9000000;
    reserve_data.b_supply = 100_0000000;
    reserve_data.d_supply = 89_0000000;
    testutils::create_reserve(&e, &pool, &underlying, &reserve_config,
&reserve_data);

    e.ledger().set(LedgerInfo {
        timestamp: 600,
        protocol_version: 22,
        sequence_number: 1234,
        network_id: Default::default(),
        base_reserve: 10,
        min_temp_entry_ttl: 10,
        min_persistent_entry_ttl: 10,
        max_entry_ttl: 3110400,
    });
    let pool_config = PoolConfig {
        oracle: Address::generate(&e),
        min_collateral: 1_0000000,
        bstop_rate: 0_2000000,
        status: 0,
        max_positions: 2,
    };

    let user_positions = Positions {
```

```
            liabilities: map![&e],
            collateral: map![&e, (0, 20_0000000)],
            supply: map![&e],
        };
        e.as_contract(&pool, || {
            storage::set_pool_config(&e, &pool_config);
            storage::set_user_positions(&e, &samwise, &user_positions);

            let mut pool = Pool::load(&e);

            let requests = vec![
                &e,
                Request {
                    request_type: RequestType::Supply as u32,
                    address: underlying.clone(),
                    amount: 10_1234567,
                },
                Request {
                    request_type: RequestType::Borrow as u32,
                    address: underlying.clone(),
                    amount: 2_0000000,
                },
                Request {
                    request_type: RequestType::Withdraw as u32,
                    address: underlying.clone(),
                    amount: 10_1234567,
                }
            ];
            let mut user = User::load(&e, &samwise);
            build_actions_from_request(&e, &mut pool, &mut user, requests);
        });
    }
```

Run `cargo test test_build_actions_from_request_borrow_errors_over_max_util` and we can see that it passes even with the `#[should_panic..]` removed, meaning the attacker has successfully taken a loan passed the util rate.

### Recommended mitigation steps

Move `reserve.require_utilization_below_max(e)` from `apply_borrow` to `validate_submit` which is the function that is called when the built transaction ends.

That way, malicious users cannot bypass this by constructing a malicious `execute_submit` transaction list.

**markus_pl10 (Script3) confirmed**

DadeKuma (validator) commented:

Bypassing max util is possible with withdraw, and it's intended to be that way. The issue seems to be that this is possible for user, even if they don't provide funds.

mootz12 (Script3) commented:

Validated this is an issue.

The root of the issue is that user's can borrow past `max_util` without actually having the funds to perform the actual `supply` invocation, or maintain the supplied position, to reduce the utilization below max util.

mootz12 (Script3) commented:

Fixed to check max util during validation portion and check 100% util during `apply_action`.

Blend mitigated:

Commit f35271b to clean up utilization checks.

**Status:** Mitigation confirmed. Full details in reports from oakcobalt, 0xAlix2, 0x007 and Testerbot.

---

## [M-15] Edge case breaks APR cap calculation and leads to excessive fee extraction from the pool

*Submitted by Sparrow*

In the fee vault's `update_rate` function, there's an edge case that affects the calculation of target growth rate when the product of `100_000 * target_apr * time_elapsed` is less than `SECONDS_PER_YEAR` (31,536,000). Due to integer division, this calculation results in 0, making the target growth rate exactly equal to `SCALAR_12` (1.0).

This issue occurs under several conditions:

- When `time_elapsed` is small (frequent updates)
- When `target_apr` is low
- Or when both conditions combine

When this happens, the target growth rate becomes 1.0 (no growth), which effectively means the protocol will take 100% of any growth as fees, contrary to the expected behavior of only taking growth above the APR cap.

### Impact

The protocol wouldn't function as intended, considering we now have the vault extracting 100% of the interest as fees, despite having a non-zero APR cap configured which directly contradicts the intended economic model and also means leak of value for the depositors since more funds are extracted from the pool (`total_b_tokens`).

Also this tends more to active reserves which would have frequent interaction with `get_reserve_vault_updated()` (which happens during deposits, withdrawals, or position queries) will trigger an update with small time intervals, causing the protocol to take all interest.

### Proof of Concept

In `fee-vault/src/reserve_vault.rs`, the problematic calculation occurs in the `update_rate` method:

reserve_vault.rs#update_rate()

```
// Target growth rate scaled in 12 decimals =
// SCALAR_12 * (target_apr / SCALAR_7) * (time_elapsed /
SECONDS_PER_YEAR) + SCALAR_12
let target_growth_rate =
    100_000 * target_apr * (time_elapsed as i128) / SECONDS_PER_YEAR +
SCALAR_12;

let target_b_rate = self
    .b_rate
    .fixed_mul_ceil(target_growth_rate, SCALAR_12)
    .unwrap();

// If the target APR wasn't reached, no fees are accrued
if target_b_rate >= new_rate {
    0
} else {
    // calculate fees
}
```

For example, with a 1% APR cap (`target_apr = 0_0100000`) and a X-second interval between updates, we get:

```
100_000 * target_apr * X = 100_000Xtarget_apr... here
`100_000Xtarget_apr` < `SECONDS_PER_YEAR`
100_000Xtarget_apr / 31,536,000 = 0 (integer division)
target_growth_rate = 0 + SCALAR_12 = SCALAR_12 (1.0)
```

This means the target growth rate becomes exactly 1.0 **(no growth)**, and when calculating `target_b_rate`:

```
target_b_rate = self.b_rate * SCALAR_12 / SCALAR_12 = self.b_rate
```

Therefore, `target_b_rate` equals the old rate with no allowed growth.

The consequence is seen in the fee calculation. Since `target_b_rate = self.b_rate`, whenever `new_rate > self.b_rate` (i.e., any positive growth), the protocol will take 100% of the growth as fees, contrary to the expected behavior of only taking growth above the APR cap:

src/reserve_vault.rs#L96-L104:

```
            // If the target APR wasn't reached, no fees are accrued
            if target_b_rate >= new_rate {
                0
            } else {
                self.total_b_tokens
                    .fixed_mul_floor(new_rate - target_b_rate, new_rate)
                    .unwrap()
            }
```

Would also be key to note that any function that calls `get_reserve_vault_updated()` which in turn calls `update_rate()` will trigger this issue under the right conditions.

### Coded POC

```
#[cfg(test)]
mod apr_capped_tests {
    use super::*;
    use crate::{
        storage::FeeMode,
        testutils::{assert_approx_eq_rel, mockpool, register_fee_vault,
EnvTestUtils},
    };
-    use soroban_sdk::{testutils::{Address as _, , Address};
+    use soroban_sdk::{testutils::{Address as _, LedgerInfo, Ledger},
Address};

### ..snip

fn test_update_rate_broken() {
        let e = Env::default();
        e.mock_all_auths();
```

```
        let init_b_rate = 1_000_000_000_000; // 1.0 with 12 decimals
        let bombadil = Address::generate(&e);

        let mock_client = &mockpool::register_mock_pool_with_b_rate(&e,
init_b_rate);

        // Set up vault with APR capped mode and a 1% APR cap
        let vault_address = register_fee_vault(
            &e,
            Some((
                bombadil.clone(),
                mock_client.address.clone(),
                true, // APR capped mode
                0_0100000, // 1% APR cap (0.01 with 7 decimals)
            )),
        );

        e.as_contract(&vault_address, || {
            // PART 1: Demonstrate the bug with a small time interval
            let mut reserve_vault = ReserveVault {
                address: Address::generate(&e),
                total_b_tokens: 1000_0000000,
                last_update_timestamp: e.ledger().timestamp(),
                total_shares: 1000_0000000,
                b_rate: init_b_rate,
                accrued_fees: 0,
            };

            // Let's set a small time interval (30 seconds)
            let initial_timestamp = e.ledger().timestamp();
            let small_time_interval = 30; // 30 seconds

            // Update the timestamp to be 30 seconds later
            e.ledger().set(LedgerInfo {
                timestamp: initial_timestamp + small_time_interval,
                protocol_version: 22,
                sequence_number: 100,
                network_id: [0; 32],
                base_reserve: 10,
                min_temp_entry_ttl: 10,
                min_persistent_entry_ttl: 10,
                max_entry_ttl: 3110400,
            });
```

```
            // CRITICAL: Set new b_rate showing 0.005% growth (annualized
to ~0.53% APR)
            // This is BELOW the 1% APR cap, so NO fees should be taken
            let new_b_rate = 1_000_050_000_000; // 1.00005 with 12
decimals (0.005% growth)
            mockpool::set_b_rate(&e, mock_client, new_b_rate);

            // Calculate what growth should be allowed by the APR cap
            // 1% APR for 30 seconds = 1% * (30 / 31536000) =
0.0000095... which is very small
            // This is approximately 0.00095% growth allowed
            // Our actual growth is 0.005%, which is well UNDER the
allowed growth
            // Therefore NO fees should be taken

            // But due to integer division in the target_growth_rate
calculation:
            // target_growth_rate = target_apr * time_elapsed /
seconds_in_year + 1e12
            // = 100_000 * 30 / 31_536_000 + 1e12
            // = 0 (due to integer division) + 1e12 = 1e12
            // target_b_rate = old_b_rate, meaning no growth is allowed

            // Before the update
            let before_total_b_tokens = reserve_vault.total_b_tokens;

            // Perform the update
            reserve_vault.update_rate(&e);

            // After the update
            let after_total_b_tokens = reserve_vault.total_b_tokens;
            let fees_taken = before_total_b_tokens -
after_total_b_tokens;

            // Calculate what the fee would be if ALL growth is taken
            // 0.005% growth on 1000_0000000 tokens = 0.05 * 1_0000000 =
0_0500000 tokens
            // With properly working APR cap, the fees should be ZERO
(since growth < APR cap)
            // But due to the precision loss, ALL growth is taken as fees

            // Assert that growth is taken as fees (bug case)
            // This is wrong because growth is under the APR cap
            assert!(fees_taken > 0);
            assert!(reserve_vault.accrued_fees > 0);
        });
```

```
    }
```

```
test reserve_vault::apr_capped_tests::test_update_rate_broken ... ok

successes:

---- reserve_vault::apr_capped_tests::test_update_rate_broken stdout ----
Writing test snapshot file for test
"reserve_vault::apr_capped_tests::test_update_rate_broken" to
"test_snapshots/reserve_vault/apr_capped_tests/test_update_rate_broken.1.
json".

successes:
    reserve_vault::apr_capped_tests::test_update_rate_broken

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 43 filtered
out; finished in 0.02s
```

### Recommended mitigation steps

Revamp the fee accrual logic to take into account lower fee rates and/or frequent updates, since we can't add a minimum time threshold for updates to prevent excessive fee extraction from very frequent small updates cause this then breaks the contracts as multiple transactions would fail on the call to `get_reserve_vault_updated()`

**markus_pl10 (Script3) confirmed**

**mootz12 (Script3) commented:**

Validated this is a finding. This is separate from S-353.

The interest math is likely not precise enough to handle small APRs / target APRs / and the minimum 5s update period, causing admin fees to not be withheld properly.

Note that this does appear to work for typical higher interest assets (USDC/EURC/etc).

**LSDan (judge) decreased severity to Medium and commented:**

Agree that it is distinct from S-353, primarily due to impact and origin of the issue. This tracks as a valid medium to me.

**mootz12 (Script3) commented:**

https://github.com/script3/fee-vault/pull/5/commits/a63d9a0c04fd7165ad7f49344faa0d60e0f85177

This won't be fixed. Unit tests were added to ensure the edge cases behaved as expected. Essentially, there always can be cases where, due to rounding, the admin gets 0 fees.

This gets more likely the less `b_tokens` that are being held by the vault, with low interest rate / low IR cases being able to scrape admin fees within a 5s interval at 1,000 `b_tokens`.

Thus, this isn't an exploitable issue, given spamming this does not cause any fund loss, only less fees get taken from the users pool of funds. If a user wants to deny an admin fees, they will be paying significantly more in TX fees than the total rounding loss for the admin, which they would only be eligible for part of.

---

## [M-16] Removal of pool from reward zone does not allow gulping emissions which were already distributed in the past

*Submitted by YouCrossTheLineAlfie, also found by Oxadrii, monrel, and rscodes*

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/emissions/manager.rs#L195

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/emissions/manager.rs#L232

### Finding description and impact

The `BackstopContract::distribute` is used to update the backstop with new emissions for all the reward zone pools.

Then, calling the `BackstopContract::gulp_emissions` with a specific pool address will gulp emissions in the ratio of 70% to the backstop and 30% to the pool.

A pool can be removed using the `BackstopContract::remove_reward` function if the threshold requirements are not met.

```
    pub fn remove_from_reward_zone(e: &Env, to_remove: Address) {
        let mut reward_zone = storage::get_reward_zone(e);

        // ensure to_add has met the minimum backstop deposit threshold
        // NOTE: "to_add" can only carry a pool balance if it is a
deployed pool from the factory
        let pool_data = load_pool_backstop_data(e, &to_remove);
        if require_pool_above_threshold(&pool_data) {           <<@ -- //
Checks if a pool is above threshold and reverts
            panic_with_error!(e, BackstopError::BadRequest);
        } else {
            remove_pool(e, &mut reward_zone, &to_remove);
            storage::set_reward_zone(e, &reward_zone);
        }
```

```
    }
```

However, while removing the pool, the `remove_pool` sets the emission index to `i128::MAX` **without** allowing to claim the emissions via `BackstopContract::gulp_emissions` which were **already** distributed.

```
    /// Remove a pool from the reward zone and set the backstop emissions
index to i128::MAX
    fn remove_pool(e: &Env, reward_zone: &mut Vec<Address>, to_remove:
&Address) {
        let to_remove_index =
reward_zone.first_index_of(to_remove.clone());
        match to_remove_index {
            Some(idx) => {
                // . . . Rest of the code . . .
                // update backstop emissions for the pool before removing
it from the reward zone
                // set emission index to i128::MAX to prevent further
emissions
                let to_remove_emis_data = storage::get_rz_emis_data(e,
&to_remove).unwrap_optimized();
                set_rz_emissions(e, &to_remove, i128::MAX,
to_remove_emis_data.accrued, false);             <<@ -- // sets i128::MAX
as the index

                reward_zone.remove(idx);
            }
            None => panic_with_error!(e,
BackstopError::InvalidRewardZoneEntry),
        }
    }
```

This would fail the `BackstopContract::gulp_emissions` call due to an overflow not allowing the rightful emissions to the pool and the backstop.

```
    pub fn gulp_emissions(e: &Env, pool: &Address) -> (i128, i128) {
        let pool_balance = storage::get_pool_balance(e, pool);

        let new_emissions = update_rz_emis_data(e, pool, true);
<<@ - // This call would revert
        // . . . Rest of the code . . .
        return (0, 0);
    }
```

The new emissions variable would overflow here as `gulp_index` is valid amount but the `emission_data.index` is set as `i128::MAX` whose difference will get multiplied by `SCALAR_14`, hence overflowing the signed 128 bit integer.

```
    pub fn update_rz_emis_data(e: &Env, pool: &Address, to_gulp: bool) ->
i128 {
        if let Some(emission_data) = storage::get_rz_emis_data(e, pool) {
            let pool_balance = storage::get_pool_balance(e, pool);
            let gulp_index = storage::get_rz_emission_index(e);
            let mut accrued = emission_data.accrued;
            if emission_data.index < gulp_index || to_gulp {
                if pool_balance.non_queued_tokens() > 0 {
                    let new_emissions = pool_balance
                        .non_queued_tokens()
                        .fixed_mul_floor(gulp_index -
emission_data.index, SCALAR_14)           <<@ -- // Overflows here
                        .unwrap_optimized();
                    accrued += new_emissions;
                    return set_rz_emissions(e, pool, gulp_index, accrued,
to_gulp);
                } else {
                    return set_rz_emissions(e, pool, gulp_index, accrued,
to_gulp);
                }
            }
        }
        return 0;
    }
```

P.S: The distribute call was made prior to the pool being under threshold or simply some other pool got higher deposits than current.

### Impact

1. Loss of funds for the users and pool as the emissions were distributed before removal from reward zone, but the gulping of them is not allowed. There's a possibility that the pool will never get added to the reward zone as pools are essentially competing with each other for a place in reward zone.
2. Temporary funds stuck for the pool if there's a chance it is added back to the reward zone.

### Proof of Concept

The below test case was added inside `blend-contracts-v2/backstop/src/emissions/manager.rs` file:

```
    #[test]
    #[should_panic]
    fn test_remove_from_rz_failing_gulp() {
        let e = Env::default();
        e.ledger().set(LedgerInfo {
            timestamp: 1713139200,
            protocol_version: 22,
            sequence_number: 0,
            network_id: Default::default(),
            base_reserve: 10,
            min_temp_entry_ttl: 10,
            min_persistent_entry_ttl: 10,
            max_entry_ttl: 3110400,
        });
        e.mock_all_auths();

        let bombadil = Address::generate(&e);
        let backstop_id = create_backstop(&e);
        let to_remove = Address::generate(&e);

        let (blnd_id, _) = create_blnd_token(&e, &backstop_id,
&bombadil);
        let (usdc_id, _) = create_usdc_token(&e, &backstop_id,
&bombadil);
        create_comet_lp_pool_with_tokens_per_share(
            &e,
            &backstop_id,
            &bombadil,
            &blnd_id,
            5_0000000,
            &usdc_id,
            0_1000000,
        );
        let mut reward_zone: Vec<Address> = vec![
            &e,
            Address::generate(&e),
            to_remove.clone(), // index 7
        ];

        e.as_contract(&backstop_id, || {
            storage::set_reward_zone(&e, &reward_zone);
```

```
            storage::set_last_distribution_time(&e, &(1713139200 - 1 * 24
* 60 * 60));
            storage::set_pool_balance(
                &e,
                &to_remove,
                &PoolBalance {
                    shares: 90_000_0000000,
                    tokens: 100_001_0000000,
                    q4w: 1_000_0000000,
                },
            );
            storage::set_pool_balance(
                &e,
                &to_remove,
                &PoolBalance {
                    shares: 35_000_0000000,
                    tokens: 40_000_0000000,
                    q4w: 1_000_0000000,
                },
            );
            storage::set_backstop_emis_data(
                &e,
                &to_remove,
                &BackstopEmissionData {
                    eps: 0_10000000000000,
                    expiration: 1713139200 + 1000,
                    index: 0,
                    last_time: 1713139200 - 12345,
                },
            );
            storage::set_rz_emis_data(&e, &to_remove, {
                &RzEmissionData {
                    index: 1234 * SCALAR_7,
                    accrued: 0,
                }
            });
            storage::set_rz_emission_index(&e, &(5678 * SCALAR_7));

remove_from_reward_zone(&e, to_remove.clone());
            let actual_rz = storage::get_reward_zone(&e);
            reward_zone.remove(1);
            assert_eq!(actual_rz.len(), 1);
            assert_eq!(actual_rz, reward_zone);
            // gulp emissions after removal
            let ( fi, si ) = gulp_emissions(&e, &to_remove.clone());
```

```
        });
    }
```

As we can infer, the function reverts as expected.

### Recommended mitigation steps

It is recommended to gulp the emissions before removing the pool:

```
    pub fn add_to_reward_zone(e: &Env, to_add: Address, to_remove:
Option<Address>) {

        // . . . Rest of the code . . .

        if MAX_RZ_SIZE > reward_zone.len() {
            // there is room in the reward zone. Add "to_add".
            reward_zone.push_front(to_add.clone());
        } else {
            match to_remove {
                None => panic_with_error!(e,
BackstopError::RewardZoneFull),
                Some(to_remove) => {
                    // Verify "to_add" has a higher backstop deposit that
"to_remove"
                    if pool_data.tokens <= storage::get_pool_balance(e,
&to_remove).tokens {
                        panic_with_error!(e,
BackstopError::InvalidRewardZoneEntry);
                    }
+                    gulp_emissions(e, &to_remove);
                    remove_pool(e, &mut reward_zone, &to_remove);
                    reward_zone.push_front(to_add.clone());
                }
            }
        }
        // . . . Rest of the code . . .
    }
```

**markus_pl10 (Script3) confirmed**

Blend mitigated:

Commit 6204dba to improve reward zone changes emissions impact.

**Status:** Mitigation confirmed. Full details in reports from Testerbot, oakcobalt, 0xAlix2 and 0x007.

## [M-17] Bad debt can be permanently blocked from being moved to backstop

*Submitted by monrel*

When a position has accumulated bad debt it is supposed to be moved to the backstop and then auctioned with a bad debt auction such that the backstop covers the debt.

This can be blocked by the owner off the position by depositing a dust amount of tokens as collateral.

A call to `pool::bad_debt()` should move the debt to the backstop but the following condition is checked.

bad_debt.rs#L26-L28

```
    if !user_state.positions.collateral.is_empty() ||
user_state.positions.liabilities.is_empty() {
        panic_with_error!(e, PoolError::BadRequest);
    }
```

There is no minimum collateral deposit so the owner of the position can add dust amount as collateral.

Not only does this block bad debt from being moved, it will also block liquidation of the dust amount since the creation of liquidation auctions with `pool:new_auction()` is not possible on dust amounts.

The owner can "release" the bad debt by depositing more collateral such that it can be liquidated again but no other actor can since deposits can only be done with the permission off the owner of each positions.

contract.rs#L427-L441

```
fn submit(
        ....
        spender.require_auth();
        if from != spender {
            from.require_auth();
        }
                ....
    }
```

The pool owner can use this position to blackmail others in the pool if the debt is substantial or to take payments from large backstop depositors to allow them to exit before the debt is

transferred.

### Proof of Concept

Here is a POC showing that a user with bad debt can permanently block it from being moved to the backstop.

▶ Details

### Recommended mitigation steps

Adding a minimum collateral allowed in the deposit is not enough since it still leaves the ability to DDOS moving the bad debt by depositing the minimum amount and forcing another liquidation auction.

We need to always check the health factor when collateral is added if liabilities exist. Disallow collateral deposits if liabilities > collateral after the deposit.

monrel (warden) commented:

I believe this has been overlooked because it has been duplicated with S-188. I will argue that this is a separate issue that is of High severity.

S-188 has identified one root cause:

1. It is possible to deposit dust to block debt removal but it requires constant front-running each time an auction is completed.

It has already been established that the above is of low severity since it requires constant front-running to block the removal briefly.

I show much more severe issue that is based on two root causes:

1. It is possible to deposit dust to block debt removal.
2. It is impossible to liquidate dust position.

By combining 1. and 2. we get the following result:

It is possible to permanently block debt removal by depositing dust amount after an auction is completed since a liquidation can not be created. This is a block that can only be lifted by the owner of that position by depositing more collateral to make it possible to create a liquidation auction. I show this in the POC of my original issue.

The revert happens it is in the following call:

src/auctions/user_liquidation_auction.rs#L127

```
let avg_cf = position_data_inc.collateral_base.fixed_div_floor(
        e,
        &position_data_inc.collateral_raw,
```

```
            &position_data_inc.scalar,
    );
```

A revert happens in `scale_mul_div_floor()` in the `i128.rs` file due to an attempt to `unwrap()` `None`. This happens because both `position_data_inc.collateral_base` and `position_data_inc.collateral_raw` have been rounded down to 0.

This report show permanent block off removal of debt which essentially means that the pools can be insolvent even when they should be solvent with the the backstop "insurance".

An attacker can use this as an attack path to blackmail pool owners. Any user with bad debt using a smart contract wallet can simply update the smart contract wallet into a contract that will ONLY allow the debt to be released if a ransom sum is deposited.

The attacker can create a closed system where the only way to remove the debt is to pay the ransom.

See this gist for a graphic explaining it.

I believe this is HIGH based on either the insolvency risk and the ability for attackers to profit on the attack based on the following C4 rules

```
- Med:Assets not at direct risk, but the function of the protocol or its
availability could be impacted, or leak value with a hypothetical attack
path with stated assumptions, but external requirements.

- High: Assets can be stolen/lost/compromised directly (or indirectly if
there is a valid attack path that does not have hand-wavy hypotheticals).
```

There is no external requirements, it can be done by any user that has accumulated bad debt.

I have demonstrated that this has a separate root cause and much more severe consequences than the duplicate, I therefore, believe that it should be a separate issue.
LSDan (judge) decreased severity to Medium and commented:

I agree this one can be treated separate, but it doesn't fit as a high risk. It requires circumstances and the impact isn't terribly high for the user. Reinstating as a valid medium.

Blend mitigated:

Commit to Initial Fix, Commit to Simplification of fix to automatically attribute bad debt if necessary after a user liquidation completes to prevent the dos window from existing.

**Status:** Mitigation confirmed. Full details in reports from 0xAlix2, 0x007 and rscodes.

---

## [M-18] Interest auctions enable inflation attacks on backstop vaults, allowing attackers to steal user deposits

*Submitted by Tricko, also found by monrel*

An attacker can exploit the interest auction mechanism in a newly created backstop vault to indirectly donate tokens, thereby manipulating the exchange rate between backstop tokens and vault shares. This manipulation can cause rounding errors in future users' deposits, enabling the attacker to steal a portion of the deposited funds.

### Finding description

Inflation attacks are a well-known threat to tokenized vaults. In this type of attack, an attacker donates tokens directly to a vault, increasing the vault's total underlying tokens. This manipulation distorts the exchange rate between tokens and shares, allowing the attacker to exploit rounding issues and steal funds from future deposits.

Backstop vaults maintain an internal accounting of their underlying tokens, so direct token transfers do not pose a risk in this codebase. However, when filling interest auctions, the pool calls the backstop's `donate()` method. This increases the internally accounted tokens of the backstop without increasing its shares, allowing an attacker to indirectly donate to the pool through interest auctions. As a result, an attacker can directly transfer tokens to a pool to falsely simulate significant interest accrual. They can then create and fill a new interest auction, using the donated funds to inflate the vault and carry out the attack. (See the PoC section below.)

Note that during deposits, the backstop's `execute_deposit` function checks whether the number of minted shares is zero and reverts the transaction in such cases. This prevents the worst-case scenario where an inflation attack could cause depositors to lose their entire deposit. **However, it does not protect against less severe cases of the attack.** Deposits larger than the donation amount will result in a nonzero number of minted shares, bypassing this check. But due to rounding errors, a substantial portion of these deposits can still be stolen by the attacker. See the PoC section below for a coded scenario demonstrating how an attacker can steal 9% from the following user deposit.

### Proof of Concept

Consider the following series of steps:

1. The attacker deposits 4 backstop tokens into a newly created backstop. **Backstop state:** (tokens: 4, shares: 4)
2. The attacker transfers `10_000 * SCALAR_7` XLM directly to the pool associated with the backstop vault.

3. The attacker calls the pool's `gulp()` method, which assigns the amount deposited in step 1 to `reserve.data.backstop_credit`.
4. The attacker initiates an interest auction on the backstop, leveraging the inflated `reserve.data.backstop_credit` due to step 2.
5. The attacker waits for one ledger.
6. The attacker fills the auction. **Backstop state:** (tokens: `960_000_0004`, shares: 4)
7. A user deposits `1200 * SCALAR_7` backstop tokens into the backstop.
8. The attacker queues for withdrawal.
9. The attacker withdraws all of his shares, leaving with a fraction of the user's deposit as profit.

**Attacker Profit:** `119_999_9998`
**User Loss:** `119_999_9998` (9% of the initial deposit)

Run the test code below for the above exemplified scenario (Copy the file contents below to a file in `blend-contracts-v2/test-suites/tests`):

▶ Details

### Recommended mitigation steps

There are multiple ways to mitigate this issue. In general, the backstop could either burn shares during its initialization or use virtual shares. Both approaches would significantly increase the cost of the attack for an attacker.

For a more targeted solution within this codebase, a restriction could be implemented to allow interest auctions to be filled only when the backstop vault already has a significant portion of shares minted. This would prevent the attack described above by ensuring that an attacker cannot exploit a newly created backstop with minimal shares.

**mootz12 (Script3) confirmed and commented:**

Validated this is a finding.

Inflation attacks are harder to pull off repeatedly in the backstop, given the withdraw queue period, and this is also mitigated slightly with prevention of zero mint scenarios.

**Blend mitigated:**

PR 48.

**Status:** Mitigation confirmed. Full details in reports from 0xAlix2, oakcobalt, Testerbot and rscodes.

*Code4rena judging staff adjusted the severity of finding [M-18], after reviewing additional context provided by the sponsor.*

---

## Low Risk and Non-Critical Issues

For this audit, 6 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **ZanyBonzy** received the top score from the judge.

*The following wardens also submitted reports: 0x007, forgebyola, Franfran, klau5, and Sparrow.*

*Note: QA report issues that were disputed have been omitted from this report, after Code4rena judging staff reviewed additional context provided by the sponsor.*

### [02] `get_market` may get dossed if there are too many reserves

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/contract.rs#L412-L421

#### Finding description and impact

`get_market` loops through amount of reserves, of which there are no ways to remove them and no limit to how many that can be set/add.

```
    fn get_market(e: Env) -> (PoolConfig, Vec<Reserve>) {
        let pool_config = storage::get_pool_config(&e);
        let res_list = storage::get_res_list(&e);
        let mut reserves = Vec::<Reserve>::new(&e);
        for res_address in res_list.iter() {
            let res = Reserve::load(&e, &pool_config, &res_address);
            reserves.push_back(res);
        }
        (pool_config, reserves)
    }
```

#### Recommended mitigation steps

Introduce a limit to how many reserves can be set. Also add a function to remove reserves.

---

### [03] Auctions cannot be created with 200 USDC interest value

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/auctions/backstop_interest_auction.rs#L63-L67

#### Finding description and impact

`create_interest_auction_data` intends that interest value is at least 200 USDC; i.e., 200 USDC or more. However, due to the <= operator in use, the function reverts if `interest_value` is 200.

```
    // Ensure that the interest value is at least 200 USDC
    if interest_value <= (200 * 10i128.pow(pool.load_price_decimals(e)))
{
        panic_with_error!(e, PoolError::InterestTooSmall);
    }
```

### Recommended mitigation steps

Change the operator to <.

**Comments from the Script3 team:**

This was addressed here to make lp token valuation pool independent for auction creation.

---

## [05] Users with 1 to 1 liability and collateral ratio can still be liquidated

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/auctions/user_liquidation_auction.rs#L50-L53

### Finding description and impact

In `create_user_liq_auction_data`, it's intended that a user has less collateral than liabilities before he can be liquidated. However, due to incorrect check, a user could have same liability as collateral (technically not being at a loss), but still be liquidated. This is because the function reverts only if `liability_base` is < `collateral_base`. If they're equal, a user can unfairly liquidated.

```
    // ensure the user has less collateral than liabilities
>   if position_data.liability_base < position_data.collateral_base {
        panic_with_error!(e, PoolError::InvalidLiquidation);
    }
```

### Recommended mitigation steps

Update the function to use <= operator instead.

**Comments from the Script3 team:**

This was addressed here to block user liquidations when liabilities equal collateral.

## [07] Consider introducing a backup oracle in case of failures

Oracles are immutable, cannot be updated once set. As a result, a potential oracle failure can topple the entire pool's ecosystem causing loss of funds to the protocol.

```
pub fn execute_initialize(
    e: &Env,
    admin: &Address,
    name: &String,
    oracle: &Address,
    bstop_rate: &u32,
    max_positions: &u32,
    min_collateral: &i128,
    backstop_address: &Address,
    blnd_id: &Address,
) {
    let pool_config = PoolConfig {
@>      oracle: oracle.clone(),
        min_collateral: *min_collateral,
        bstop_rate: *bstop_rate,
        status: 6,
        max_positions: *max_positions,
    };
    require_valid_pool_config(e, &pool_config);

    storage::set_admin(e, admin);
    storage::set_name(e, name);
    storage::set_backstop(e, backstop_address);
    storage::set_pool_config(e, &pool_config);
    storage::set_blnd_token(e, blnd_id);
}
```

### Recommended mitigation steps

Reconsider oracle immutability or add the option for a backup oracle.

## [08] Absence of a flashfee will dos a execute with flashfee fxns

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/pool/submit.rs#L114-L121

### Finding description and impact

FlashLoanClient calls `exec_op` passing in 0 as the flash fee. If the client charges a flashfee, it will be unusable since no fee is being passed in.

```
    FlashLoanClient::new(&e, &flash_loan.contract).exec_op(
        &from,
        &flash_loan.asset,
        &flash_loan.amount,
@>      &0,
    );
```

### Recommended mitigation steps

Introduce a function to query potential clients' fees, and pass the return value into the function.

---

## [10] No function to get already set name

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/storage.rs#L230-L234

### Finding description and impact

Contract allows setting pool names, but there is no equivalent function to get them. Integrators may not be able to query the name of a pool if needed.

```
pub fn set_name(e: &Env, name: &String) {
    e.storage()
        .instance()
        .set::<Symbol, String>(&Symbol::new(e, NAME_KEY), name);
}
```

### Recommended mitigation steps

Add a function to return pool name.

---

## [11] Emit an event after dropping

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/contract.rs#L298-L305

### Finding description and impact

Other operations in backstop contract emit an event upon completion except `drop` which fails to an event even after performing a significant state change.

```
    fn drop(e: Env) {
        let mut drop_list = storage::get_drop_list(&e);
        let backfilled_emissions = storage::get_backfill_emissions(&e);
        drop_list.push_back((e.current_contract_address(),
backfilled_emissions));
        let emitter_client = EmitterClient::new(&e,
&storage::get_emitter(&e));
        emitter_client.drop(&drop_list)
    }
```

---

## [12] Adjust incorrect parameter naming issue in `set_emitter`

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/storage.rs#L128-L132

### Finding description and impact

`set_emitter` allows setting emitter, but the parameter name is set as `pool_factory_id`. This works because both parameters are addresses, but in other cases could cause a parameter mismatch issue.

```
@>  pub fn set_emitter(e: &Env, pool_factory_id: &Address) {
      e.storage()
          .instance()
}
```

### Recommended mitigation steps

Change `pool_factory_id` to emitter id instead.

**Comments from the Script3 team:**

This was addressed here.

---

## [13] Update incorrect comment on `set_emitter`

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/backstop/src/storage.rs#L128-L132

### Finding description and impact

`set_emitter` should set emitter and not pool factory.

```
>/// Set the pool factory
///
/// ### Arguments
>/// * `pool_factory_id` - The ID of the pool factory
pub fn set_emitter(e: &Env, pool_factory_id: &Address) {
    e.storage()
        .instance()
        .set::<Symbol, Address>(&Symbol::new(e, EMITTER_KEY),
pool_factory_id);
}
```

### Recommended mitigation steps

Update comments.

---

## [18] Contrary to whitepaper, backstop rate can be changed

https://docs.blend.capital/blend-whitepaper#owned-pools

https://github.com/code-423n4/2025-02-blend/blob/f23b3260763488f365ef6a95bfb139c95b0ed0f9/blend-contracts-v2/pool/src/contract.rs#L326-L327

### Finding description and impact

According to the whitepaper, the backstop rate should be immutable and set upon deployment.

Owned pools are isolated lending pools where a delegated address can modify pool state and most pool parameters. Notably, they cannot modify the oracle contract address parameter or the backstop take rate parameter. This restriction prevents excessive damage to users by malicious or compromised pool owners.

But in our implementation, the `update_pool` allows setting a new backstop rate.

```
pub fn execute_update_pool(
```

```
    e: &Env,
    backstop_take_rate: u32,
    max_positions: u32,
    min_collateral: i128,
) {
    let mut pool_config = storage::get_pool_config(e);
    pool_config.bstop_rate = backstop_take_rate;
    pool_config.max_positions = max_positions;
    pool_config.min_collateral = min_collateral;

    require_valid_pool_config(e, &pool_config);
    storage::set_pool_config(e, &pool_config);
}
```

---

# Mitigation Review

## Introduction

Following the C4 audit, 6 wardens (0x007, oakcobalt, Testerbot, rscodes and a_kalout and ali_shehab of team 0xAlix2) reviewed the mitigations implemented by the Script3 team. Additional details can be found within the C4 Blend Mitigation Review repository.

## Mitigation Review - Scope & Summary

The wardens confirmed the mitigations for all in-scope findings except for M-01, where the finding was not mitigated. They also surfaced one new issue of low severity. The table below provides details regarding the status of each in-scope vulnerability from the original audit, followed by full details on the new issue and the in-scope vulnerability that was not fully mitigated.

| ORIGINAL ISSUE | STATUS | MITIGATION URL |
|---|---|---|
| H-01 | 🟢 Mitigation confirmed | Commit e4ed914 |
| H-03 | 🟢 Mitigation confirmed | PR 48 |
| M-01 | 🟡 Unmitigated - Follow-up not reviewed | Commit e4ed914 |
| M-02 | 🟢 Mitigation confirmed | Commit f35271b |
| M-03 | 🟢 Mitigation confirmed | PR 48 |

| ORIGINAL ISSUE | STATUS | MITIGATION URL |
|---|---|---|
| M-04 | 🟢 Mitigation confirmed | Commit fc6a2af |
| M-05 | 🟢 Mitigation confirmed | Commit 77373e3 |
| M-06 | 🟢 Mitigation confirmed | Commit 77373e3 |
| M-08 | 🟢 Mitigation confirmed | Commit 6204dba |
| M-09 | 🟢 Mitigation confirmed | Commit 59acbc9; Improvements |
| M-10 | 🟢 Mitigation confirmed | Commit f0296cf |
| M-11 | 🟢 Mitigation confirmed | Commit a63d9a0 |
| M-12 | 🟢 Mitigation confirmed | Commit 6204dba |
| M-13 | 🟢 Mitigation confirmed | Commit b3e5af4 |
| M-14 | 🟢 Mitigation confirmed | Commit f35271b |
| M-16 | 🟢 Mitigation confirmed | Commit 6204dba |
| M-17 | 🟢 Mitigation confirmed | Commit to Initial Fix, Commit to Simplification of fix |
| M-18 | 🟢 Mitigation confirmed | PR 48 |
| ADD-01 | 🟢 Mitigation confirmed | PR 50 |

## M-01 Unmitigated

*Submitted by 0xAlix2, also found by oakcobalt*

**Original issue:** https://code4rena.com/evaluate/2025-02-blend-v2-audit-certora-formal-verification/submissions/F-5

### Finding description and impact

The issue was that the flash loan flow was missing a borrow-enabled check, allowing users to bypass that check. If borrowing is disabled, the "default" borrow functionality doesn't work, but it works through flash loan.

This increases the risk of protocol insolvency.

Mitigation here.

A pool borrow-enabled check is added in `execute_submit_with_flash_loan`, here:

```
pool.require_action_allowed(e, RequestType::Borrow as u32); // <-----
Here
let mut reserve = pool.load_reserve(e, &flash_loan.asset, true);
```

However, this is missing the other part of this, which is the reserve's validation, that is reported in S-308.

### Proof of Concept

Add the following in `pool/src/pool/submit.rs`:

```
#[test]
fn test_flash_loan_disabled_reserve() {
    let e = Env::default();
    e.cost_estimate().budget().reset_unlimited();
    e.mock_all_auths_allowing_non_root_auth();

    e.ledger().set(LedgerInfo {
        timestamp: 600,
        protocol_version: 22,
        sequence_number: 1234,
        network_id: Default::default(),
        base_reserve: 10,
        min_temp_entry_ttl: 10,
        min_persistent_entry_ttl: 10,
        max_entry_ttl: 3110400,
    });

    let bombadil = Address::generate(&e);
    let samwise = Address::generate(&e);
    let pool = testutils::create_pool(&e);
    let (oracle, oracle_client) = testutils::create_mock_oracle(&e);

    let (flash_loan_receiver, _) =
testutils::create_flashloan_receiver(&e);

    let (underlying_0, _) = testutils::create_token_contract(&e,
&bombadil);
    let (mut reserve_config, mut reserve_data) =
testutils::default_reserve_meta();
    reserve_config.max_util = 9500000;
```

```
    reserve_data.b_supply = 100_0000000;
    reserve_data.d_supply = 50_0000000;
    testutils::create_reserve(&e, &pool, &underlying_0, &reserve_config,
&reserve_data);

    let (underlying_1, underlying_1_client) =
testutils::create_token_contract(&e, &bombadil);
    let (reserve_config, reserve_data) =
testutils::default_reserve_meta();
    testutils::create_reserve(&e, &pool, &underlying_1, &reserve_config,
&reserve_data);

    oracle_client.set_data(
        &bombadil,
        &Asset::Other(Symbol::new(&e, "USD")),
        &vec![
            &e,
            Asset::Stellar(underlying_0.clone()),
            Asset::Stellar(underlying_1.clone()),
        ],
        &7,
        &300,
    );
    oracle_client.set_price_stable(&vec![&e, 1_0000000, 5_0000000]);

    e.as_contract(&pool, || {
        storage::set_pool_config(
            &e,
            &PoolConfig {
                oracle,
                min_collateral: 1_0000000,
                bstop_rate: 0_1000000,
                status: 0,
                max_positions: 4,
            },
        );

        underlying_1_client.mint(&samwise, &25_0000000);
        underlying_1_client.approve(&samwise, &pool, &100_0000000,
&10000);

        let mut reserve_0_config = storage::get_res_config(&e,
&underlying_0);
        reserve_0_config.enabled = false;
        storage::set_res_config(&e, &underlying_0, &reserve_0_config);
```

```
        let positions = execute_submit_with_flash_loan(
            &e,
            &samwise,
            FlashLoan {
                contract: flash_loan_receiver,
                asset: underlying_0.clone(),
                amount: 25_0000000,
            },
            vec![
                &e,
                Request {
                    request_type: RequestType::SupplyCollateral as u32,
                    address: underlying_1.clone(),
                    amount: 25_0000000,
                },
            ],
        );

        assert_eq!(positions.liabilities.len(), 1);
        assert!(positions.liabilities.get_unchecked(0) > 0);
    });
}
```

### Recommended mitigation steps

```
    pub fn execute_submit_with_flash_loan(
        e: &Env,
        from: &Address,
        flash_loan: FlashLoan,
        requests: Vec<Request>,
    ) -> Positions {
        if from == &e.current_contract_address() {
            panic_with_error!(e, &PoolError::BadRequest);
        }
        let mut pool = Pool::load(e);
        let mut from_state = User::load(e, from);

        let prev_positions_count =
from_state.positions.effective_count();

        // note: we add the flash loan liabilities before processing the
other
        // requests.
        {
            pool.require_action_allowed(e, RequestType::Borrow as u32);
```

```
            let mut reserve = pool.load_reserve(e, &flash_loan.asset,
true);
+           reserve.require_action_allowed(e, RequestType::Borrow as
u32);
            let d_tokens_minted = reserve.to_d_token_up(e,
flash_loan.amount);
            from_state.add_liabilities(e, &mut reserve, d_tokens_minted);
            reserve.require_utilization_below_100(e);

            pool.cache_reserve(reserve);

            PoolEvents::flash_loan(
                e,
                flash_loan.asset.clone(),
                from.clone(),
                flash_loan.contract.clone(),
                flash_loan.amount,
                d_tokens_minted,
            );
        }

        // ... snip ...

    }
```

### Links to affected code

submit.rs#L86-L94

**mootz12 (Script3) commented:**

Confirmed this was missed. It has been addressed here.

---

## Invalid division by 0 validation in `convert_to_shares`, DoSing the backstop pool

*Submitted by 0xAlix2*

**Severity: Low**

NB: This was disputed in the previous contest; however, we still believe this is an actual issue that needs to be fixed, even if the likelihood of it is low.

### Finding description and impact

When users deposit BLND:USDC LP tokens into the backstop pools, they receive shares in return. These shares are calculated using a standard ERC4626-style formula:

```
shares = (deposit amount × total shares) / total token deposits
```

Similarly, when users redeem shares, the inverse is used to calculate how many tokens they receive back — potentially with profit. To avoid division-by-zero errors, special care must be taken when either total shares or total tokens are zero.

This logic is implemented in the `PoolBalance` struct:

```
pub fn convert_to_shares(&self, tokens: i128) -> i128 {
@>  if self.shares == 0 {
        return tokens;
    }

    tokens
        .fixed_mul_floor(self.shares, self.tokens)
        .unwrap_optimized()
}

pub fn convert_to_tokens(&self, shares: i128) -> i128 {
    if self.shares == 0 {
        return shares;
    }

    shares
        .fixed_mul_floor(self.tokens, self.shares)
        .unwrap_optimized()
}
```

However, note the issue in `convert_to_shares`: the check if `self.shares == 0` is not fully accurate. Since the formula divides by total tokens, the zero check should be on `self.tokens` — not `self.shares`. Otherwise, it could lead to incorrect results or division-by-zero in edge cases.

Leading to DoS of both future deposits and claiming previous rewards.

Let's take the following scenario:

A pool got into bad debt; this bad debt is >= the tokens deposited into the corresponding backstop vault, and this bad debt gets auctioned. That auction is filled, and 100% of the corresponding backstop vault deposits are drawn. After that, the pool gets back into a healthy state, new suppliers come in, and borrowing/repaying is back as normal.

However, at that point, the corresponding backstop vault is completely DoSed, it doesn't allow any new deposits, and it also blocks users who previously earned some rewards to be

claimed; they both use `convert_to_shares`.

Importantly, this DoS is not caused by any explicit logic blocking deposits or claims. Instead, it results from a faulty division-by-zero check, which incorrectly prevents further interaction with the vault.

As a result, the pool continues operating without an active backstop vault, leaving it unprotected against future bad debt events.

NB: In normal 4626 vaults, for example, Solmate, it is enough to check against shares because there's no way to decrease the tokens without burning shares, unlike the backstop vault case, when `draw` is called.

### Proof of Concept

Add the following test in `test-suites/tests/test_backstop_rz_changes.rs`:

```
#[test]
fn test_wrong_division_by_zero_check() {
    let fixture = create_fixture_with_data(false);
    let bstop_token = &fixture.lp;
    let sam = Address::generate(&fixture.env);
    let pool_fixture = &fixture.pools[0];

    fixture.tokens[TokenIndex::BLND].mint(&sam, &
(125_001_000_0000_0000_000_000 * SCALAR_7)); // 10 BLND per LP token
    fixture.tokens[TokenIndex::BLND].approve(&sam, &bstop_token.address,
&i128::MAX, &99999);
    fixture.tokens[TokenIndex::USDC].mint(&sam, &
(3_126_000_0000_0000_000_000 * SCALAR_7)); // 0.25 USDC per LP token
    fixture.tokens[TokenIndex::USDC].approve(&sam, &bstop_token.address,
&i128::MAX, &99999);
    bstop_token.join_pool(
        &(2 * 12_500 * SCALAR_7),
        &vec![
            &fixture.env,
            125_001_000_0000_0000_000 * SCALAR_7,
            3_126_000_0000_0000_000 * SCALAR_7,
        ],
        &sam,
    );
    fixture
        .backstop
        .deposit(&sam, &pool_fixture.pool.address, &(12500 * SCALAR_7));

    fixture.jump(60 * 60 * 24 * 21);
```

```
    fixture.emitter.distribute();
    fixture.backstop.distribute();
    pool_fixture.pool.gulp_emissions();

    fixture.backstop.draw(
        &pool_fixture.pool.address,
        &fixture
            .backstop
            .pool_data(&pool_fixture.pool.address)
            .tokens,
        &sam,
    );

    assert_eq!(
        fixture
            .backstop
            .pool_data(&pool_fixture.pool.address)
            .tokens,
        0
    );

    let deposit_res =
        fixture
            .backstop
            .try_deposit(&sam, &pool_fixture.pool.address, &(12500 *
SCALAR_7));

    // Reverts, division by zero
    assert!(deposit_res.is_err());

    let claim_res = fixture.backstop.try_claim(
        &sam,
        &vec![&fixture.env, pool_fixture.pool.address.clone()],
        &0,
    );

    // Reverts, division by zero
    assert!(claim_res.is_err());
}
```

### Recommended mitigation steps

```
    impl PoolBalance {
        /// Convert a token balance to a share balance based on the
current pool state
```

```
         ///
         /// ### Arguments
         /// * `tokens` - the token balance to convert
         pub fn convert_to_shares(&self, tokens: i128) -> i128 {
-            if self.shares == 0 {
+            if self.tokens == 0 {
                 return tokens;
             }

             tokens
                 .fixed_mul_floor(self.shares, self.tokens)
                 .unwrap_optimized()
         }

         // ... snip ...
     }
```

### Links to affected code

pool.rs#L139-L141

**mootz12 (Script3) commented:**

I believe there was an issue along these lines in the original review. Not sure where.

The finding is correct that a backstop locks once the backstop is completely drained. However, a backstop being locked when completely drained is not a bad thing.

This mainly has to do with the fact that unless all shares a burnt, the next depositor will lose likely all of their deposit. If we mint 1-1, it's likely the backstop still has `>10k-100k` shares outstanding during a complete bad debt scenario, Thus, the depositing user would end up donating tokens to the other share holders.

There really isn't a way to resolve this safely. If the pool keeps being used, an interest auction will occur that will donate tokens back into the vault, allowing it to function again.

There is an issue in what this function returns in the case where shares `> 0` and tokens `== 0`. It should return 0 if tokens `== 0`, given the pool backstop is out of tokens.

**a_kalout (warden) commented**

I want to clarify some points please. At first, the sponsor mentioned that this is a non-issue and that it's "okay" to have it. However, after some discussions in a PT, I can share if needed for transparency, they concluded that this is indeed a valid issue that needs to be fixed (I know that's not enough to have this judged as valid). This is mentioned by the sponsor:

There is an issue in what this function returns in the case where shares `> 0` and tokens `== 0`. It should return 0 if tokens `== 0`, given the pool backstop is out of tokens.

Where they mentioned that it should return 0 in that case, I don't agree with that; I believe tokens should be returned as shares because the healthiness of a backstop is measured by the price of the shares in assets and not by the minted amount (replying to "If we mint 1-1, it's likely the backstop still has `>10k-100k` shares outstanding during a complete bad debt scenario"). The depositor's assets will still be diluted in the pool.

Regardless of the applied mitigation, the current implementation doesn't do both and instead reverts, causing transitory DoS (which I respectfully believe guarantees med severity).

Having this is an invalid, according to this, means that the report didn't provide any value to the sponsor, which is not the case here, as a fix for this was applied here.
LSDan (judge) commented:

Valid low - I don't see any real impact in terms of funds lost or protocol function. That would be required to make this a medium.

**Comments from the Script3 team:**

This has been addressed here to check the reserve status on flash loan.

---

## Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.