Formal Verification Report: Blend Protocol v2 Backstop

- Competition: https://code4rena.com/audits/2025-02-blend-v2-audit-certora-formal-verification
- Repository: https://github.com/code-423n4/2025-02-blend-fv
- Latest Commit Hash: 6b803fa
- Scope: blend-contracts-v2/backstop
- Date: February 2025
- Author: @alexzoid_eth
- Certora Prover version: 7.26.0

Table of Contents

Formal Verification Methodology

Certora Formal Verification (FV) provides mathematical proofs of smart contract correctness by verifying code against a formal specification. It complements techniques like testing and fuzzing, which can only sometimes detect bugs based on predefined properties. In contrast, Certora FV examines all possible states and execution paths in a contract.

Simply put, the formal verification process involves crafting properties (similar to writing tests) in native RUST language and submitting them alongside compiled programs to a remote prover. This prover essentially transforms the program bytecode and rules into a mathematical model and determines the validity of rules.

Types of Properties

When constructing properties in formal verification, we mainly deal with two types: **Invariants** and **Rules**. Invariants are implemented in parametric style (one property for each external function) with `parametric_rule!()` macros.

The structure of parametric rule:

- Initialize ghost storage from rule parameters (this a hack to reduce complexity of storage infractions)
- Assume realistic timestamp
- Assume valid state invariants hold
- Log all storage variables
- Execute external function
- Log all storage variables again

Invariants

- Conditions that MUST **always remain true** throughout the contract's lifecycle. Implemented with `invariant_rule!()`. Similar to parametric rules, but check the property hold with `cvlr_assert!()` macros.
- Process:

    1. Define an initial condition for the contract's state.

    2. Execute an external function.

    3. Confirm the invariant still holds after execution.

- Example: "The total shares MUST always equal the sum of all user shares."
- Use Case: Ensures **Valid State** properties - critical state constraints that MUST never be violated.
- Feature: Proven invariants can be reused in other properties.

Rules

- Flexible checks for specific behaviors or conditions.
- Structure:

    1. Setup: Set valid state assumptions (e.g., "user balance is non-zero") with `init_verification!()` macros.
    2. Execution: Simulate contract behavior by calling external functions.
    3. Verification:
        - Use `cvlr_assert!()` to check if a condition is **always true** (e.g., "balance never goes negative").
        - Use `cvlr_satisfy!()` to verify a condition is **reachable** (e.g., "a user can withdraw funds").
- Example: "A withdrawal decreases the user's balance."
- Use Case: Verifies a broad range of properties, from simple state changes to complex business logic.

Verification Process

The process is divided into two stages: **Setup** and **Crafting Properties**.

Setup

This stage prepares the contract and prover for verification. Use conditional source code compilation with `just features`.

- Resolve external contract calls, declare mocks in `mocks` and `summaries` directory (with `certora_token_mock`, `certora_pool_factory_mock` and `certora_emission_summarized` features)
- Simplify complex operations (mirroring storage r/w operations into ghosts variables with `certora_storage_ghost` feature, vector interactions with `certora_vec_q4w`) to reduce timeouts.
- Prove **Valid State properties** (invariants) as a foundation for further checks.

Crafting Properties

This stage defines and implements the properties:

- Write properties in **plain English** for clarity.
- Categorize properties by purpose (e.g., Valid State, Variable Transition). Some of them are implemented in parametric style (`valid_state.rs`, `state_trans.rs`, `sanity.rs`, `isolation.rs`), while others (`integrity_*.rs`, `high_level.rs`) as regular rules.
- Use proven valid state invariants as assumptions in **Rules** for efficiency.

Assumptions

Assumptions simplify the verification process and are classified as **Safe** or **Unsafe**. Safe assumptions are backed by valid state invariants or required by the environment. Unsafe made to reduce complexity, potentially limiting coverage.

Safe Assumptions

Timestamp Constraints

- Block timestamps are always non-zero (`e.ledger().timestamp() > 0`)

Valid State Invariants

These invariants are proven to always hold and can be safely assumed:

Non-negative Value Invariants:

- `valid_state_nonnegative_pb_shares`: Pool balance shares are non-negative
- `valid_state_nonnegative_pb_tokens`: Pool balance tokens are non-negative
- `valid_state_nonnegative_pb_q4w`: Pool balance Q4W amounts are non-negative
- `valid_state_nonnegative_ub_shares`: User balance shares are non-negative
- `valid_state_nonnegative_ub_q4w_amount`: User Q4W entry amounts are non-negative

Pool Balance Invariants:

- `valid_state_pb_q4w_leq_shares`: Pool Q4W total does not exceed pool shares

User Balance Invariants:

- `valid_state_ub_shares_plus_q4w_sum_eq_pb_shares`: User shares + Q4W amounts equal pool shares
- `valid_state_ub_q4w_sum_eq_pb_q4w`: Sum of user Q4W amounts equals pool Q4W total
- `valid_state_ub_q4w_expiration`: Q4W entry expiration times do not exceed timestamp + Q4W_LOCK_TIME
- `valid_state_ub_q4w_exp_implies_amount`: Q4W entries with expiration have non-zero amounts

General State Invariants:

- `valid_state_user_not_pool`: User addresses cannot be pool or contract addresses (zero balance enforced)
- `valid_state_pool_from_factory`: Only factory-deployed pools can have non-zero balances

Unsafe Assumptions

Mocks and Summaries

- Token contracts mocked with `certora_token_mock` feature
- Pool factory mocked with `certora_pool_factory_mock` feature
- Emission calculations summarized with `certora_emission_summarized` feature

Loop Unrolling

- Vector iterations limited to 2 iterations (`loop_iter = 2` in configs)

Verification Properties

The verification properties are categorized into the following types:

1. **Valid State (VS)**: System state invariants that MUST always hold
2. **State Transitions (ST)**: Rules governing state changes during operations
3. **Isolation (ISO)**: Properties verifying operation independence and non-interference
4. **Sanity (SA)**: Basic reachability and functionality checks
5. **High Level (HL)**: Complex business logic and protocol-specific rules
6. **Integrity (INT)**: Properties ensuring data consistency and correctness

Each job status linked to a corresponding run in the dashboard with a specific status:

- ✅ completed successfully
- ⚠️ reached global timeout
- ❌ violated

Valid State

The states define the possible values that the system's variables can take. These invariants ensure the backstop contract maintains consistency at all times.

All valid state properties are stored in valid_state.rs and ✅ passed verification successfully.

| Source	Invariant	Description	Caught mutations
VS-01	valid_state_nonnegative_pb_shares	Pool balance shares are non-negative	-
VS-02	valid_state_nonnegative_pb_tokens	Pool balance tokens are non-negative	❌deposit_3, ❌fundmanagement_2, ❌fundmanagement_4
VS-03	valid_state_nonnegative_pb_q4w	Pool balance Q4W amounts are non-negative	❌pool_4
VS-04	valid_state_nonnegative_ub_shares	User balance shares are non-negative	-
VS-05	valid_state_nonnegative_ub_q4w_amount	User Q4W entry amounts are non-negative	-
VS-06	valid_state_pb_q4w_leq_shares	Pool Q4W total must not exceed pool shares	❌pool_2, ❌withdraw_1
VS-07	valid_state_ub_shares_plus_q4w_sum_eq_pb_shares	User shares + Q4W amounts must equal pool shares	❌deposit_0, ❌deposit_1, ❌pool_1, ❌user_0, ❌user_1, ❌user_3, ❌withdraw_2
VS-08	valid_state_ub_q4w_sum_eq_pb_q4w	Sum of user Q4W amounts must equal pool Q4W total	❌pool_2, ❌pool_4, ❌user_3, ❌withdraw_0, ❌withdraw_1
VS-09	valid_state_ub_q4w_expiration	Q4W entry expiration times do not exceed timestamp + Q4W_LOCK_TIME	-
VS-10	valid_state_ub_q4w_exp_implies_amount	Q4W entries with expiration must have non-zero amounts	-
VS-11	valid_state_user_not_pool	User addresses cannot be pool or contract addresses	❌fund_management_1

Source	Invariant	Description	Caught mutations
VS-12	valid_state_pool_from_factory	Only factory-deployed pools can have non-zero balances	-

State Transitions

These properties verify that state changes occur correctly during contract operations.

All state transition properties are stored in state_trans.rs and ✅ passed verification successfully.

Source	Rule	Description	Caught mutations
ST-01	state_trans_pb_shares_tokens_directional_change	Pool shares and tokens change in same direction	❌deposit_3, ❌pool_0, ❌pool_3, ❌withdraw_3
ST-02	state_trans_pb_q4w_consistency	Pool Q4W changes are consistent with operations	❌user_1, ❌user_3, ❌withdraw_1
ST-03	state_trans_ub_shares_increase_consistency	User balance consistency when shares increase	-
ST-04	state_trans_ub_shares_decrease_consistency	User balance consistency when shares decrease	❌pool_4
ST-05	state_trans_ub_q4w_amount_consistency	User Q4W amount changes are properly tracked	❌pool_1, ❌pool_2, ❌pool_4, ❌user_1, ❌user_3

Isolation

Properties verifying that operations on different pools and users are properly isolated.

All isolation properties are stored in isolation.rs and passed verification: ✅ pool isolation and ✅ user isolation successfully.

Source	Rule	Description	Caught mutations
ISO-01	isolation_pool	Operations on one pool don't affect others	-
ISO-02	isolation_user	Operations by one user don't affect others	-

Sanity

Basic checks ensuring contract functions remain accessible and operational.

All sanity properties are stored in sanity.rs and ✅ passed verification successfully.

Source	Rule	Description	Caught mutations
SA-01	sanity	All external functions remain callable under valid state	-

High Level

Complex business logic and protocol-specific properties.

All high-level properties are stored in high_level.rs and ✅ passed verification successfully.

Source	Rule	Description	Caught mutations
HL-01	high_level_deposit_returns_converted_shares	Deposit returns correct share conversion	-
HL-02	high_level_withdrawal_expiration_enforced	Withdrawals can't happen before expiration	-
HL-03	high_level_share_token_initial_conversion	1:1 conversion when pool is empty	-
HL-04	high_level_share_token_conversion	Consistent token/share conversion rates	-

Integrity

Properties ensuring data integrity and calculation correctness throughout operations.

All integrity properties ✅ passed verification successfully.

Balance Integrity

All balance integrity properties are stored in integrity_balance.rs.

Source	Rule	Description	Caught mutations
INT-01	integrity_balance_deposit	Deposits correctly update balances	❌deposit_1, ❌deposit_2, ❌deposit_3, ❌pool_3, ❌user_0
INT-02	integrity_balance_withdraw	Withdrawals correctly update balances	❌pool_0, ❌pool_1, ❌pool_2, ❌user_3
INT-03	integrity_balance_queue_withdrawal	Queue withdrawal correctly updates balances	❌pool_4, ❌user_1, ❌withdraw_1
INT-04	integrity_balance_dequeue_withdrawal	Dequeue withdrawal correctly updates balances	❌user_0, ❌withdraw_0, ❌withdraw_2

Source	Rule	Description	Caught mutations
INT-05	integrity_balance_donate	Donations correctly update balances	❌fundmanagement_3, ❌fundmanagement_4, ❌pool_3
INT-06	integrity_balance_draw	Draw operations correctly update balances	❌fundmanagement_0, ❌pool_0
INT-07	integrity_balance_load_pool_backstop_data	Data loading doesn't change state	-

Emission Integrity

All emission integrity properties are stored in integrity_emission.rs.

Source	Rule	Description	Caught mutations
INT-08	integrity_emission_deposit	Emission state correct during deposit	-
INT-09	integrity_emission_withdraw	Emission state correct during withdraw	-
INT-10	integrity_emission_queue_withdrawal	Emission state correct during queue withdrawal	-
INT-11	integrity_emission_dequeue_withdrawal	Emission state correct during dequeue withdrawal	-
INT-12	integrity_emission_donate	Emission state correct during donate	-
INT-13	integrity_emission_draw	Emission state correct during draw	-

Token Integrity

All token integrity properties are stored in integrity_token.rs.

Source	Rule	Description	Caught mutations
INT-14	integrity_token_deposit	Token transfers correct during deposit	-
INT-15	integrity_token_withdraw	Token transfers correct during withdraw	-
INT-16	integrity_token_donate	Token transfers correct during donate	-

Source	Rule	Description	Caught mutations
INT-17	integrity_token_draw	Token transfers correct during draw	-
INT-18	integrity_token_queue_withdrawal	No token transfers during queue withdrawal	-
INT-19	integrity_token_dequeue_withdrawal	No token transfers during dequeue withdrawal	-

Manual Mutations Testing

This section documents the manual mutations from the Certora FV contest applied to the five key backstop contract files. Each caught mutation is tested against specific rules to verify that our specifications correctly detect the introduced bugs.

Deposit

mutations/deposit/deposit_0.rs

Comments out the pool balance deposit operation while keeping user balance update, breaking balance consistency.

```
    let to_mint = pool_balance.convert_to_shares(amount);
    if to_mint <= 0 {
        panic_with_error!(e, &BackstopError::InvalidShareMintAmount);
    }
    // pool_balance.deposit(amount, to_mint); MUTANT
    user_balance.add_shares(to_mint);

    storage::set_pool_balance(e, pool_address, &pool_balance);
```

Caught by:

- ❌ valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_deposit

mutations/deposit/deposit_1.rs

Comments out the user balance share addition while keeping pool balance update, creating inconsistent state.

```
    let to_mint = pool_balance.convert_to_shares(amount);
    if to_mint <= 0 {
        panic_with_error!(e, &BackstopError::InvalidShareMintAmount);
    }
    pool_balance.deposit(amount, to_mint);
    // user_balance.add_shares(to_mint); MUTANT

    storage::set_pool_balance(e, pool_address, &pool_balance);
```

Caught by:

- ❌ valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_deposit
- ❌ integrity_balance_deposit

mutations/deposit/deposit_2.rs

Removes validation check for zero or negative share amounts, allowing invalid deposits.

```
    let to_mint = pool_balance.convert_to_shares(amount);
    // if to_mint <= 0 { MUTANT
    //     panic_with_error!(e, &BackstopError::InvalidShareMintAmount);
    // }
    pool_balance.deposit(amount, to_mint);
    user_balance.add_shares(to_mint);
```

Caught by:

- ❌ integrity_balance_deposit

mutations/deposit/deposit_3.rs

Removes input validation for negative amounts, allowing deposits with negative values.

```
pub fn execute_deposit(e: &Env, from: &Address, pool_address: &Address, amount: i128) ->
i128 {
    // require_nonnegative(e, amount); MUTANT
    if from == pool_address || from == &e.current_contract_address() {
        panic_with_error!(e, &BackstopError::BadRequest)
    }
```

Caught by:

- ❌ valid_state_nonnegative_pb_tokens_execute_deposit
- ❌ state_trans_pb_shares_tokens_directional_change_execute_deposit
- ❌ integrity_balance_deposit

Fund Management

mutations/fundmanagement/fund_management_0.rs

Inserts spurious withdrawal operation with zero amounts, potentially affecting balance tracking.

```
    let mut pool_balance = storage::get_pool_balance(e, pool_address);

    pool_balance.withdraw(e, 0, 0); // MUTANT
    storage::set_pool_balance(e, pool_address, &pool_balance);

    #[cfg(feature = "certora_token_mock")] // @note changed
```

Caught by:

- ❌ [integrity_balance_draw](#)

mutations/fundmanagement/fund_management_1.rs

Removes input validation for negative amounts in draw operations, allowing invalid withdrawals.

```
pub fn execute_draw(e: &Env, pool_address: &Address, amount: i128, to: &Address) {
    // require_nonnegative(e, amount); MUTANT

    let mut pool_balance = storage::get_pool_balance(e, pool_address);
```

Caught by:

- ❌ [valid_state_user_not_pool_execute_draw](#)

mutations/fundmanagement/fund_management_2.rs

Removes input validation for negative amounts in donate operations, allowing invalid donations.

```
pub fn execute_donate(e: &Env, from: &Address, pool_address: &Address, amount: i128) {
    // require_nonnegative(e, amount); MUTANT
    if from == pool_address || from == &e.current_contract_address() {
        panic_with_error!(e, &BackstopError::BadRequest)
    }
```

Caught by:

- ❌ [valid_state_nonnegative_pb_tokens_execute_donate](#)

mutations/fundmanagement/fund_management_3.rs

Replaces proper amount deposit with zero values, breaking balance tracking in donations.

```
    }

    pool_balance.deposit(0, 0); // MUTANT
    storage::set_pool_balance(e, pool_address, &pool_balance);
}
```

Caught by:

- ❌ [integrity_balance_donate](#)

mutations/fundmanagement/fund_management_4.rs

Deposits incorrect token amount (amount - 1) instead of the full amount, creating balance discrepancies.

```
    }

    pool_balance.deposit(amount - 1, 0); // MUTANT
    storage::set_pool_balance(e, pool_address, &pool_balance);
}
```

Caught by:

- ❌ valid_state_nonnegative_pb_tokens_execute_donate
- ❌ integrity_balance_donate

Pool

mutations/pool/pool_0.rs

Comments out the token balance reduction during withdrawal while keeping share updates, breaking token-share consistency.

```
if tokens > self.tokens || shares > self.shares || shares > self.q4w {
    panic_with_error!(e, BackstopError::InsufficientFunds);
}
// self.tokens -= tokens; MUTANT
self.shares -= shares;
self.q4w -= shares;
```

Caught by:

- ❌ state_trans_pb_shares_tokens_directional_change_execute_withdraw
- ❌ integrity_balance_draw
- ❌ integrity_balance_withdraw

mutations/pool/pool_1.rs

Comments out the share balance reduction during withdrawal while keeping token and queue updates, breaking share accounting.

```
if tokens > self.tokens || shares > self.shares || shares > self.q4w {
    panic_with_error!(e, BackstopError::InsufficientFunds);
}
self.tokens -= tokens;
// self.shares -= shares; MUTANT
self.q4w -= shares;
```

Caught by:

- ❌ valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_withdraw
- ❌ state_trans_ub_q4w_amount_consistency_execute_withdraw
- ❌ integrity_balance_withdraw

mutations/pool/pool_2.rs

Comments out the queue-for-withdrawal balance reduction during withdrawal, breaking queue consistency.

```
        if tokens > self.tokens || shares > self.shares || shares > self.q4w {
            panic_with_error!(e, BackstopError::InsufficientFunds);
        }
        self.tokens -= tokens;
        self.shares -= shares;
        // self.q4w -= shares; MUTANT
```

Caught by:

- ❌ valid_state_ub_q4w_sum_eq_pb_q4w_execute_withdraw
- ❌ valid_state_pb_q4w_leq_shares_execute_withdraw
- ❌ state_trans_ub_q4w_amount_consistency_execute_withdraw
- ❌ integrity_balance_withdraw

mutations/pool/pool_3.rs

Comments out the token balance increase during deposit while keeping share updates, breaking token-share consistency.

```
    pub fn deposit(&mut self, tokens: i128, shares: i128) {
        // self.tokens += tokens; MUTANT
        self.shares += shares;
    }
```

Caught by:

- ❌ state_trans_pb_shares_tokens_directional_change_execute_deposit
- ❌ integrity_balance_deposit
- ❌ integrity_balance_donate

mutations/pool/pool_4.rs

Changes the queue-for-withdrawal operation from addition to subtraction, causing negative balances.

```
    pub fn queue_for_withdraw(&mut self, shares: i128) {
        self.q4w -= shares; // MUTANT changed + to -
    }
```

Caught by:

- ❌ valid_state_nonnegative_pb_q4w_execute_queue_withdrawal
- ❌ valid_state_ub_q4w_sum_eq_pb_q4w_execute_queue_withdrawal
- ❌ state_trans_ub_q4w_amount_consistency_execute_queue_withdrawal
- ❌ state_trans_ub_shares_decrease_consistency_execute_queue_withdrawal
- ❌ integrity_balance_queue_withdrawal

User

mutations/user/user_0.rs

Replaces the share addition parameter with zero, preventing user balance updates.

```
    pub fn add_shares(&mut self, to_add: i128) {
        self.shares += 0; // MUTANT
    }
```

Caught by:

- ❌ valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_deposit
- ❌ valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_dequeue_withdrawal
- ❌ integrity_balance_deposit
- ❌ integrity_balance_dequeue_withdrawal

mutations/user/user_1.rs

Changes user share reduction to addition during queue operation, causing incorrect balance calculations.

```
        self.shares = self.shares + to_q; // MUTANT

        // user has enough tokens to withdrawal, add Q4W
        let new_q4w = Q4W {
```

Caught by:

- ❌ valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_queue_withdrawal
- ❌ state_trans_pb_q4w_consistency_execute_queue_withdrawal
- ❌ state_trans_ub_q4w_amount_consistency_execute_queue_withdrawal
- ❌ integrity_balance_queue_withdrawal

mutations/user/user_3.rs

Changes withdrawal amount comparison from greater-than-or-equal to less-than, causing incorrect queue processing logic.

```
            if cur_q4w.exp <= e.ledger().timestamp() {
                if cur_q4w.amount < left_to_withdraw { // MUTANT
                    // last record we need to update, but the q4w should remain
```

Caught by:

- ❌ valid_state_ub_q4w_sum_eq_pb_q4w_execute_withdraw
- ❌ valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_withdraw
- ❌ state_trans_pb_q4w_consistency_execute_withdraw
- ❌ state_trans_ub_q4w_amount_consistency_execute_withdraw

- ❌ [integrity_balance_withdraw](#)

Withdrawal

mutations/withdraw/withdraw_0.rs

Comments out the user balance storage update during dequeue, preventing balance persistence.

```
    user_balance.dequeue_shares(e, amount);
    user_balance.add_shares(amount);
    pool_balance.dequeue_q4w(e, amount);

    // storage::set_user_balance(e, pool_address, from, &user_balance); MUTANT
    storage::set_pool_balance(e, pool_address, &pool_balance);
```

Caught by:

- ❌ [valid_state_ub_q4w_sum_eq_pb_q4w_execute_dequeue_withdrawal](#)
- ❌ [integrity_balance_dequeue_withdrawal](#)

mutations/withdraw/withdraw_1.rs

Comments out the user queue operation during withdrawal queuing, breaking user-pool queue consistency.

```
    // user_balance.queue_shares_for_withdrawal(e, amount); MUTANT
    pool_balance.queue_for_withdraw(amount);

    storage::set_user_balance(e, pool_address, from, &user_balance);
```

Caught by:

- ❌ [valid_state_pb_q4w_leq_shares_execute_queue_withdrawal](#)
- ❌ [valid_state_ub_q4w_sum_eq_pb_q4w_execute_queue_withdrawal](#)
- ❌ [state_trans_pb_q4w_consistency_execute_queue_withdrawal](#)
- ❌ [integrity_balance_queue_withdrawal](#)

mutations/withdraw/withdraw_2.rs

Comments out the user share addition during dequeue, preventing share reallocation to user.

```
    user_balance.dequeue_shares(e, amount);
    // user_balance.add_shares(amount); MUTANT
    pool_balance.dequeue_q4w(e, amount);

    storage::set_user_balance(e, pool_address, from, &user_balance);
```

Caught by:

- ❌ [valid_state_ub_shares_plus_q4w_sum_eq_pb_shares_execute_dequeue_withdrawal](#)
- ❌ [integrity_balance_dequeue_withdrawal](#)

[mutations/withdraw/withdraw_3.rs](#)

Comments out the zero-amount withdrawal validation, allowing invalid withdrawals to proceed.

```
    let to_return = pool_balance.convert_to_tokens(amount);
    // if to_return == 0 { MUTANT
    //     panic_with_error!(e, &BackstopError::InvalidTokenWithdrawAmount);
    // }
    pool_balance.withdraw(e, to_return, amount);
```

Caught by:

- ❌ [state_trans_pb_shares_tokens_directional_change_execute_withdraw](#)

Real Bug Finding

Zero-amount Withdrawal Queue Entry

Finding description and impact:

The [execute_queue_withdrawal](#) function allows users to queue zero-amount entries for withdrawal, they provide no actual withdrawal value and consume limited queue slots (`MAX_Q4W_SIZE = 20`). This can lead to additional transaction overhead to dequeue zero entries one by one and a misleading queue state.

Proof of Concept:

This behavior [violates](#) the formal verification invariant `valid_state_ub_q4w_exp_implies_amount`, which specifies that `Q4W` entries with non-zero expiration times always should have non-zero amounts.

```
// If a Q4W entry has a non-zero expiration time, it must have a non-zero amount
pub fn valid_state_ub_q4w_exp_implies_amount(
    e: Env,
    pool: Address,
    user: Address
) -> bool {
    let ub: UserBalance = storage::get_user_balance(&e, &pool, &user);

    if ub.q4w.len() != 0 {
        let entry0 = ub.q4w.get(0).unwrap_optimized();
        // If expiration is set (non-zero), amount must also be set (non-zero)
        if entry0.exp > 0 && entry0.amount == 0 {
            return false;
        }
    }

    true
}
```

Recommended mitigation steps:

```
diff --git a/blend-contracts-v2/backstop/src/backstop/withdrawal.rs b/blend-contracts-
v2/backstop/src/backstop/withdrawal.rs
index e3664f0..47e6e1b 100644
--- a/blend-contracts-v2/backstop/src/backstop/withdrawal.rs
+++ b/blend-contracts-v2/backstop/src/backstop/withdrawal.rs
@@ -21,6 +21,10 @@ pub fn execute_queue_withdrawal(
 ) -> Q4W {
     require_nonnegative(e, amount);

+    if amount == 0 {
+        panic_with_error!(e, BackstopError::InternalError);
+    }
+
     let mut pool_balance = storage::get_pool_balance(e, pool_address);
     let mut user_balance = storage::get_user_balance(e, pool_address, from);
```

FV rule [passed](#) after the fix.

Setup and Execution Instructions

Certora Prover Installation

For step-by-step installation steps refer to this setup [tutorial](#).

Verification Execution

1. Build the backstop contract with Certora features:

```
cd blend-contracts-v2/backstop/confs
just build
```

2. Run a specific verification:

```
certoraSorobanProver <config_file>.conf
```

3. Run all verifications:

```
./run_conf.sh
```

4. Run verifications matching a pattern:

```
./run_conf.sh <pattern>
```