code4rena

Echelon

Smart Contract Security Assessment

Version 1.0

Audit dates: Dec 18 — Jan 09, 2025

Audited by: Victor Magnum
Peakbolt

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Echelon

Echelon

2.2 Scope

Repository	EchelonMarket/echelon-modules
Commit Hash	d5b0dd26bd7b6e7c510ae75fc8824aa9e79733df

2.3 Audit Timeline

DATE	EVENT
Dec 18, 2024	Audit start
Jan 09, 2025	Audit end
Jan 14, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	6
Low Risk	4
Informational	0
Total Issues	10

3. Findings Summary

ID	DESCRIPTION	STATUS
M-1	Rounding errors in `unstake_boosted()` will prevent full loan repayment and withdrawal	Resolved
M-2	Borrow cap should be checked against `amount_post_origination_fee` in `borrow_internal()`	Resolved

M-3	Set interest rate model should call `accrue_market_interest()` first	Resolved
M-4	`set_interest_fee_bps()` will apply new fee bps retroactively	Resolved
M-5	Protocol could incur further losses for bad debts with paused markets	Resolved
M-6	Borrowers could be unfairly liquidated when protocol/market is un-paused	Resolved
L-1	`vetoken` and `dividend_distributor` could be DoS if locked asset ownership change	Acknowledged
L-2	Potential unbounded loop in `claimable_internal()`	Acknowledged
L-3	`account_borrowable_coins_rate_limited()` should also consider borrowing cap and supply	Resolved
L-4	`set_market_origination_fee_bps()` should check `origination_fee_bps <= BPS_BASE`	Resolved

4. Findings

4.1 Medium Risk

A total of 6 medium risk findings were identified.

[M-1] Rounding errors in `unstake_boosted()` will prevent full loan repayment and withdrawal

Severity: Medium	Status: Resolved

Context:

- farming.move#L553-L567

Description: With `BoostedFarming`, the `stake_amount` for the farming pools will be adjusted by a boost multiplier based on the account's balance of the ve tokens.

And `stake()`/`unstake()` will now call `apply_boost_multiplier()` to update the `stake_amount` in all the existing pools with the account's current boost multiplier. The `stake_amount` is divided by the previous boost multiplier to obtain the original `stake_amount` before applying the new boost multiplier.

However, the precision loss from the division in `apply_boost_multiplier()`, will cause the `stake_amount` to be rounded down. This will will cause `unstake()` to fail when the unstake amount is greater than `stake_amount`. This prevents the user from performing a full repayment or withdrawal, causing the protocol to accumulate large amount of micro loans that cannot be closed. These will eventually manifest into bad debts that has to be absorbed by protocol.

Suppose the following scenario,

1. In Pool A, Alice has an original `stake_amount == 999` that is the same as her loan amount for Token A, and boost multiplier at `1`.
2. Now Alice's multiplier is increased to `1.11`.
3. Alice does does a full loan repayment for Token A, which will indirectly call `unstake_boosted(999)`.
4. If we follow the code below for `unstake_boosted()`, `boosted_amount = 1.11 * 999 = 1108.89 = 1109` due to the `decode_round_up()`.
5. After that `apply_boost_multiplier()` will adjust the `stake_amount = 999 * 1.11/1 = 1108.89 = 1108` due to the `decode_round_down()`.
6. Then when `unstake_internal(boosted_amount)` is called, it will fail as it tries to unstake `1109` but the `stake_amount` has been adjusted to `1008`. This cause the full loan

repayment to fail.

```
    fun unstake_boosted(account_addr: address, farming_identifier:
String, amount: u64) acquires Farming, BoostedFarming, Staker {
        // 1) calculate boosted_amount
        let curr_boost_multiplier = boost_multiplier(account_addr);
>>>     let boosted_amount =
fixed_point64::decode_round_up(fixed_point64::mul(curr_boost_multiplier,
amount));

        // 2) unstake
>>>     apply_boost_multiplier(account_addr, curr_boost_multiplier);
>>>     unstake_internal(account_addr, farming_identifier,
boosted_amount);

        // 3) update boost_multiplier
        let boosted_farming = borrow_global_mut<BoostedFarming>
(package::package_address());
        smart_table::upsert(&mut boosted_farming.boost_multipliers,
account_addr, curr_boost_multiplier);
    }

fun apply_boost_multiplier(account_addr: address, curr_boost_multiplier:
FixedPoint64) acquires Staker, BoostedFarming, Farming {
        // If user does not exist yet, there is no need to apply boost
factor
        if (!exists<Staker>(account_addr)) { return };

        // 1) calculate boost_multiplier_to_apply
        let boosted_farming = borrow_global<BoostedFarming>
(package::package_address());
        let prev_boost_multiplier =
*smart_table::borrow_with_default(&boosted_farming.boost_multipliers,
account_addr, &fixed_point64::one());
        let boost_multiplier_to_apply =
fixed_point64::div_fp(curr_boost_multiplier, prev_boost_multiplier);

        // 2) update all farming pools the user stakes to
        let staker = borrow_global_mut<Staker>(account_addr);
        vector::for_each(simple_map::keys(&staker.user_pools),
|farming_identifier| {
            let stake_amount = simple_map::borrow(&mut staker.user_pools,
&farming_identifier).stake_amount;
            let new_stake_amount =
fixed_point64::decode_round_down(fixed_point64::mul(boost_multiplier_to_a
pply, stake_amount));
```

```
            unstake_internal(account_addr, farming_identifier,
stake_amount);
            stake_internal(account_addr, farming_identifier,
new_stake_amount);
        })
    }
```

Recommendation:

This can be resolved as follows,

1. Storing the unboosted `stake_amount` in a new storage variable e.g. `original_stake_amount` instead of deriving it from dividing by `prev_boost_multiplier` to prevent rounding down from precision loss.
2. Use `fixed_point64::decode_round_down()` instead of `fixed_point64::decode_round_up()` in `unstake_boosted()` to prevent rounding errors.

Echelon: Fixed in [PR-226]

Zenith: Verified. Also included the init functions to migrate the `original_stake_amount` into `BoostedFarming` on the first interaction (stake/unstake/claim).

[M-2] Borrow cap should be checked against `amount_post_origination_fee` in `borrow_internal()`

Severity: Medium	Status: Resolved

Context:

- lending.move#L2244

Description: In `borrow_internal()`, the borrow cap should be checked against `amount_post_origination_fee` and not `amount`.

That is because the origination fee is part of the user's loan and not paid upfront, so it should be considered as borrowed coins/assets and count towards the borrow cap. This is evident from the code that adds `amount_post_origination_fee` to `market.total_liability`.

This issue will cause users to borrow more than the borrow cap.

```
    fun borrow_internal(account: &signer, market_obj: Object<Market>,
amount: u64) acquires Market, Vault, Lending, JumpInterestRateModel,
CoinInfo, FungibleAssetInfo, PauseFlag, MarketRateLimit,
MarketOriginationFee  {
       ...
        // check borrow cap
        let market_origination_fee =
borrow_market_origination_fee(market_obj);
        let market = borrow_mut_market(market_obj);
        assert!(market.total_cash >= amount,
ERR_LENDING_INSUFFICIENT_UNDERLYING_BALANCE);
>>>     assert!(market.total_liability + amount <= market.borrow_cap,
ERR_LENDING_MARKET_BORROW_CAP_EXCEEDED);

        // withdraw coin from market, update market total_liability,
total_cash
        let origination_fee_amount = mul_div_rounded(amount,
market_origination_fee.value_bps, BPS_BASE, true);
        let amount_post_origination_fee = amount +
origination_fee_amount;
        market.total_liability = market.total_liability +
amount_post_origination_fee;
        market.total_cash = market.total_cash - amount;
        market.total_reserve = market.total_reserve +
origination_fee_amount;
        let total_liability_ = market.total_liability;
        let total_cash_ = market.total_cash;
```

Recommendation: This can be resolved as follows,

```
    fun borrow_internal(account: &signer, market_obj: Object<Market>,
amount: u64) acquires Market, Vault, Lending, JumpInterestRateModel,
CoinInfo, FungibleAssetInfo, PauseFlag, MarketRateLimit,
MarketOriginationFee  {
       ...
        // check borrow cap
        let market_origination_fee =
borrow_market_origination_fee(market_obj);
        let market = borrow_mut_market(market_obj);
        assert!(market.total_cash >= amount,
ERR_LENDING_INSUFFICIENT_UNDERLYING_BALANCE);
-       assert!(market.total_liability + amount <= market.borrow_cap,
ERR_LENDING_MARKET_BORROW_CAP_EXCEEDED);

        // withdraw coin from market, update market total_liability,
total_cash
        let origination_fee_amount = mul_div_rounded(amount,
market_origination_fee.value_bps, BPS_BASE, true);
        let amount_post_origination_fee = amount +
origination_fee_amount;
+       assert!(market.total_liability + amount_post_origination_fee <=
market.borrow_cap, ERR_LENDING_MARKET_BORROW_CAP_EXCEEDED);
        market.total_liability = market.total_liability +
amount_post_origination_fee;
        market.total_cash = market.total_cash - amount;
        market.total_reserve = market.total_reserve +
origination_fee_amount;
        let total_liability_ = market.total_liability;
        let total_cash_ = market.total_cash;
```

Echelon: Fixed in PR-223

Zenith: Verified

[M-3] Set interest rate model should call `accrue_market_interest()` first

Severity: Medium | Status: Resolved

Context:

- lending.move#L555-L574
- isolated_lending.move#L473-L492

Description: `set_market_jump_interest_rate_model()` and `set_pair_jump_interest_rate_model()` allows the manager to change the interest rate parameters for the specified market.

However, it did not call `accrue_market_interest()` to accrue all the pending interest based on the existing interest rate model before updating to the new model.

When `accrue_market_interest()` is next called, this issue will cause the new interest rate model to be applied retroactively on the pending interest before the interest rate model update.

```
    public entry fun set_market_jump_interest_rate_model(manager:
&signer, market_obj: Object<Market>, base_rate_bps: u64, multiplier_bps:
u64, jump_multiplier_bps: u64, utilization_kink_bps: u64) acquires
Market, JumpInterestRateModel {
        assert!(manager::is_manager(manager), ERR_LENDING_UNAUTHORIZED);
        assert!(utilization_kink_bps <= BPS_BASE,
ERR_LENDING_INVALID_UTILIZATION_KINK_BPS);
        assert_market_exists(market_obj);

        let market_signer = market_signer(market_obj);
        let market = borrow_mut_market(market_obj);

        // remove old interest rate model struct

remove_old_interest_rate_model(signer::address_of(&market_signer),
market.interest_rate_model_type);

        // create new interest rate model struct
        market.interest_rate_model_type = INTEREST_RATE_MODEL_TYPE_JUMP;
        move_to(&market_signer, JumpInterestRateModel {
            base_rate_bps,
            multiplier_bps,
            jump_multiplier_bps,
            utilization_kink_bps
        });
    }
```

```
    public entry fun set_pair_jump_interest_rate_model(manager: &signer,
pair_obj: Object<Pair>, base_rate_bps: u64, multiplier_bps: u64,
jump_multiplier_bps: u64, utilization_kink_bps: u64) acquires Pair,
JumpInterestRateModel {
        assert!(manager::is_manager(manager),
ERR_ISOLATED_LENDING_UNAUTHORIZED);
        assert!(utilization_kink_bps <= BPS_BASE,
ERR_ISOLATED_LENDING_INVALID_UTILIZATION_KINK_BPS);
        assert_pair_exists(pair_obj);

        let pair_signer = pair_signer(pair_obj);
        let pair = borrow_mut_pair(pair_obj);

        // remove old interest rate model struct
        remove_old_interest_rate_model(signer::address_of(pair_signer),
pair.interest_rate_model_type);

        // create new interest rate model struct
        pair.interest_rate_model_type = INTEREST_RATE_MODEL_TYPE_JUMP;
        move_to(pair_signer, JumpInterestRateModel {
            base_rate_bps,
            multiplier_bps,
            jump_multiplier_bps,
            utilization_kink_bps
        });
    }
```

Recommendation: Call `accrue_market_interest()` in `set_market_jump_interest_rate_model()` and `accrue_pair_interest()` in `set_pair_jump_interest_rate_model()` before updating the interest rate model.

Echelon: Fixed in PR-220

Zenith: Verified.

[M-4] `set_interest_fee_bps()` will apply new fee bps retroactively

Severity: Medium	Status: Resolved

Context:

- lending.move#L492-L499
- isolated_lending.move#L398-L405

Description: `interest_fee_bps` determines the protocol fee amount (in bps) on the interest earned and `set_interest_fee_bps()` can be called by the manager to change it.

However, when the `set_interest_fee_bps()` is called, it fails to accrue the pending interest for the markets.

This will cause the next accrual to use the new `interest_fee_bps` on pending interest that should had been applied with the previous `interest_fee_bps`.

```
    public entry fun set_interest_fee_bps(manager: &signer,
interest_fee_bps: u64) acquires Lending {
        assert!(manager::is_manager(manager), ERR_LENDING_UNAUTHORIZED);
        assert!(interest_fee_bps <= BPS_BASE,
ERR_LENDING_INVALID_INTEREST_FEE_BPS);

        // update interest_fee_bps
        let lending = borrow_global_mut<Lending>
(package::package_address());
        lending.interest_fee_bps = interest_fee_bps;
    }
```

```
    public entry fun set_interest_fee_bps(manager: &signer,
interest_fee_bps: u64) acquires IsolatedLending {
        assert!(manager::is_manager(manager),
ERR_ISOLATED_LENDING_UNAUTHORIZED);
        assert!(interest_fee_bps <= BPS_BASE,
ERR_ISOLATED_LENDING_INVALID_INTEREST_FEE_BPS);

        // update interest_fee_bps
        let isolated_lending = borrow_global_mut<IsolatedLending>
(package::package_address());
        isolated_lending.interest_fee_bps = interest_fee_bps;
    }
```

Recommendation: In `set_interest_fee_bps()`, accrue the pending interest for all the markets to apply the existing `interest_fee_bps` before updating with the new `interest_fee_bps`.

Echelon: Fixed in PR-219

Zenith: Verified.

[M-5] Protocol could incur further losses for bad debts with paused markets

Severity: Medium	Status: Resolved

Context:

- lending.move#L994

Description: `clear_bad_debt()` in `lending.move` allows the manager to seize the collateral of bad debts and write it off as a loss for the protocol.

However, it does not allow bad debts to be cleared for markets that are paused.

That is undesirable for the following reasons,

1 The protocol will incur more losses if the paused collateral value drops further and it cannot be cleared. That is plausible for volatile collateral and possibly the reason for the bad debt (which was not liquidated in time). 2. It prevents clearing the bad debts timely, allowing it to accrue more interests, which the protocol have to absorb later. 3. It could zero out vault's liability without seizing the collateral shares if only the collateral market is paused. This could make the vault healthy again and it cannot be called by `clear_bad_debt()` again to seize those collateral shares. 4. These bad debts are still accruing rewards in the farming pools for the borrowers, and there are currently no mechanism to stop it till the bad debts are cleared.

Point 3 has the worst impact as the users will have their liabilities zero out (loan written off) without seizing their collateral (for paused markets). That means they can withdraw those collaterals when it is un-paused, without repaying the loan anymore.

Recommendation: It is recommended to allow `clear_bad_debt()` to be used for paused markets as well.

Echelon: Fixed in PR-217

Zenith: Verified.

[M-6] Borrowers could be unfairly liquidated when protocol/market is un-paused

Severity: Medium	Status: Resolved

Context:

- lending.move#L2297-L2324
- isolated_lending.move#L1528-L1533

Description: Both `lending.move` and `isolated_lending.move` allows the manager to pause the specific markets or the entire lending protocol.

However, when the market/protocol is un-paused, liquidations are immediately allowed on the loans.

This will cause certain borrowers to be unfairly liquidated because interest continues to accrue on their loans, but they are unable to bring their loan to a healthy level due to the pause on supply and repay as well.

```
    fun liquidate_internal(liquidator: &signer, borrower_addr: address,
borrow_market_obj: Object<Market>, collateral_market_obj: Object<Market>,
repay_amount: u64, min_shares_out: u64) acquires Vault, Market, Lending,
JumpInterestRateModel, CoinInfo, FungibleAssetInfo,
MarketLiquidationThreshold, EmodeLiquidationThreshold,
MarketLiquidationIncentive, PauseFlag {
        // check lending status: not paused
        assert!(!paused(), ERR_LENDING_PAUSED);

        // check collateral market and borrow market: not paused
        let borrow_market = borrow_market(borrow_market_obj);
        let collateral_market = borrow_market(collateral_market_obj);
        assert!(!borrow_market.paused, ERR_LENDING_MARKET_PAUSED);
        assert!(!collateral_market.paused, ERR_LENDING_MARKET_PAUSED);
```

```
    fun liquidate_internal(liquidator: &signer, borrower_addr: address,
pair_obj: Object<Pair>, repay_shares: u64, min_amount_out: u64): (u64,
u64) acquires IsolatedLending, Pair, AccountPositions,
JumpInterestRateModel, CoinInfo, FungibleAssetInfo {
        assert_pair_exists(pair_obj);

        // check pair status: not paused
        let pair = borrow_pair(pair_obj);
        assert!(!pair.paused, ERR_ISOLATED_LENDING_PAIR_PAUSED);
```

Recommendation: This can be fixed by implementing a liquidation grace period (e.g. 1hr) when unpausing the market/protocol. This is similar to Aave's Liquidation grace sentinel.

During the liquidation grace period, liquidations continue to be paused, to give borrowers time to top-up collaterals or repay their loans. After that, liquidations will be allowed as normal.

Note: The liquidation grace period can also be used in the future if there is a need to adjust parameters (e.g. lower liquidation_threshold_bps, removing collateral from emode) that could make certain loans instantly liquidatable.

Echelon: Fixed in PR-225

Zenith: Verified. Resolved with a market liquidation pause flag that allows liquidation to continue be paused when unpausing protocol/market, giving users grace period to repay/top-up collateral before resuming liquidation.

4.2 Low Risk

A total of 4 low risk findings were identified.

[L-1] `vetoken` and `dividend_distributor` could be DoS if locked asset ownership change

Severity: Low	Status: Acknowledged

Context:

- dividend_distributor.move#L236
- dividend_distributor.move#L199
- dividend_distributor.move#L216
- vetoken.move#L703
- vetoken.move#L698

Description: In `dividend_distributor` and `vetoken`, there are multiple use of `create_object_address()` to derive the `dividend_distributor_address` and `vetoken_info_address` using the locked asset's `object::owner()`.

However, if the locked asset's ownership is transferred to a new owner, `create_object_address()` will fail to derive `dividend_distributor_address` and `vetoken_info_address` correctly as they were created based on the address of the initial owner. That will lead to a DoS of both `vetoken` and `dividend_distributor`, preventing users from claiming their dividend.

```
    fun claimable_internal(account_addr: address, lock_metadata:
Object<Metadata>, dividend_metadata: Object<Metadata>): (u64, u64)
acquires DividendDistributor {
        let total_claimable = 0;
        let seed = dividend_distributor_seed(lock_metadata,
dividend_metadata);
>>>     let dividend_distributor_address =
object::create_object_address(&object::owner(lock_metadata), seed);
```

```
    #[view]
    public fun vetoken_info_address(metadata: Object<Metadata>): address
{
>>>     object::create_object_address(&object::owner(metadata),
vetoken_info_seed(metadata))
```

```
        }
```

Recommendation: This can be acknowledged by ensuring that the locked asset owner does not change. That is because both `dividend_distributor` and `vetoken` can only be initialized by the locked asset's owner, which means that the ownership is under the protocol's control.

Alternatively, this can be resolved by storing both `dividend_distributor_address` and `vetoken_info_address` in the global storage and used instead of deriving it with `create_object_address`.

Echelon: Acknowledged, wont fix.

Zenith: Acknowledged by the client as locked asset ownership will not change.

[L-2] Potential unbounded loop in `claimable_internal()`

Severity: Low	Status: Acknowledged

Context:

- dividend_distributor.move#L233-L262

Description: The `distributor.epoch_dividend` vector tracks the amount of total dividend distributed for each epoch. That means the vector length correspond to the number of past and upcoming epochs.

When a new user wish to claim the dividend on their vetoken, `claimable_internal()` will loop from index `0` of `distributor.epoch_dividend` to the latest claimable index.

An issue could occur if the epoch duration is too low, which increases the length of `distributor.epoch_dividend` over time. That will cause new users to spend large amount of gas to perform a dividend claim for their vetoken. The claim could also fail if it exceeds the gas limits.

```
    fun claimable_internal(account_addr: address, lock_metadata:
Object<Metadata>, dividend_metadata: Object<Metadata>): (u64, u64)
acquires DividendDistributor {
        let total_claimable = 0;
        let seed = dividend_distributor_seed(lock_metadata,
dividend_metadata);
        let dividend_distributor_address =
object::create_object_address(&object::owner(lock_metadata), seed);
        let distributor = borrow_global<DividendDistributor>
(dividend_distributor_address);
        let now_epoch = vetoken::now_epoch(lock_metadata);

        let i =
*smart_table::borrow_with_default(&distributor.next_claimable,
account_addr, &0);
        let n = smart_vector::length(&distributor.epoch_dividend);
>>>     while (i < n) {
            let record =
smart_vector::borrow(&distributor.epoch_dividend, i);
            // Only past epochs are claimable
            if (record.epoch >= now_epoch) {
                break
            };
            let epoch_balance =
vetoken::unnormalized_past_balance(account_addr, lock_metadata,
record.epoch);
            if (epoch_balance == 0) {
                i = i + 1;
                continue
            };

            let epoch_total_supply =
vetoken::unnormalized_past_total_supply(lock_metadata, record.epoch);
            let claimable = (((record.dividend_amount as u128) *
epoch_balance / epoch_total_supply) as u64);
            total_claimable = total_claimable + claimable;

            i = i + 1;
        };

        (total_claimable, i)
    }
```

Recommendation: This can be acknowledged by ensuring that epoch duration is not too short. Also consider running a unit test with the worst case number of epochs for the loop to verify that the gas required will not exceed the gas limit.

Echelon: Acknowledged, wont fix.

Zenith: Unlikely to encounter issues as frequency of dividend distribution is low (biweekly) and at most weekly.

[L-3] `account_borrowable_coins_rate_limited()` should also consider borrowing cap and supply

Severity: Low   Status: Resolved

Context:

- lending.move#L1934-L1940

Description: The function `account_borrowable_coins_rate_limited()` provides the borrowable coins for the account, taking into consideration of the market rate limit for borrowing.

As this function serves to compute the borrowable coins with the market level borrowing adjustment (rate limit), it should also take into considerations the borrowing cap for the market and the supply, to give an accurate value.

This allows the user and frontend to be given the precise figure of borrowable coins before the user try to perform the borrow, preventing a possible failed transaction.

```
    #[view]
    public fun account_borrowable_coins_rate_limited(account_addr:
address, market_obj: Object<Market>): u64 acquires Vault, Market,
Lending, JumpInterestRateModel, CoinInfo, FungibleAssetInfo,
MarketRateLimit, MarketOriginationFee {
        let amount_to_borrow = account_borrowable_coins(account_addr,
market_obj);

        let rate_limit_left = (rate_limiter::remaining(&mut
borrow_mut_market_rate_limit(market_obj).borrow,
timestamp::now_seconds()) as u64);
        math64::min(amount_to_borrow, rate_limit_left)
    }
```

Recommendation: Consider incorporating the borrowing cap adjustment and supply to return the true amount of coins that can borrowed by the account.

Echelon: Fixed in PR-231

Zenith: Verified. Resolved with updated `account_borrowable_coins()` that accounts for rate limiting, supply and borrowing cap.

[L-4] `set_market_origination_fee_bps()` should check `origination_fee_bps <= BPS_BASE`

Severity: Low Status: Resolved

Context:

- lending.move#L620-L631

Description: `set_market_origination_fee_bps()` should validate that the `origination_fee_bps` is less than `BPS_BASE`.

```
  public entry fun set_market_origination_fee_bps(manager: &signer,
market_obj: Object<Market>, origination_fee_bps: u64) acquires Market,
MarketOriginationFee {
        assert!(manager::is_manager(manager), ERR_LENDING_UNAUTHORIZED);
        assert_market_exists(market_obj);

        // Create or update market origination fee
        if (!exists<MarketOriginationFee>
(object::object_address(&market_obj))) {
            move_to(&market_signer(market_obj), MarketOriginationFee {
value_bps: origination_fee_bps });
        } else {
            let market_origination_fee =
borrow_mut_market_origination_fee(market_obj);
            market_origination_fee.value_bps = origination_fee_bps;
        }
    }
```

Recommendation:

```
  public entry fun set_market_origination_fee_bps(manager: &signer,
market_obj: Object<Market>, origination_fee_bps: u64) acquires Market,
MarketOriginationFee {
        assert!(manager::is_manager(manager), ERR_LENDING_UNAUTHORIZED);
+         assert!(origination_fee_bps<= BPS_BASE,
ERR_LENDING_INVALID_MARKET_ORIGINATION_FEE_BPS);
        assert_market_exists(market_obj);

        // Create or update market origination fee
        if (!exists<MarketOriginationFee>
(object::object_address(&market_obj))) {
```

```
            move_to(&market_signer(market_obj), MarketOriginationFee {
value_bps: origination_fee_bps });
        } else {
            let market_origination_fee =
borrow_mut_market_origination_fee(market_obj);
            market_origination_fee.value_bps = origination_fee_bps;
        }
    }
```

Echelon: Fixed in PR-222

Zenith: Verified. Resolved by checking against `MAX_ORIGINATION_FEE_BPS` that is below `BPS_BASE`.