Echelon

Security Assessment

May 8th, 2025 — Prepared by OtterSec

Andreas Mantzoutas andreas@osec.io

Bartłomiej Wierzbiński dark@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Echelon engaged OtterSec to assess the `echelon-modules` program. This assessment was conducted between April 1st and April 16th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 17 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability where the current implementation allows unauthorized creation of a primary store at the package address or for the pool's asset, which may prevent the addition of farming rewards and block pool creation by making a valid primary store unavailable (OS-ECH-ADV-02). Additionally, a missing solvency check allows users to withdraw supply shares even when their position has bad debt, risking protocol insolvency (OS-ECH-ADV-01), and a lack of functionality to unfreeze coin stores post a freeze operation may result in permanent lockups (OS-ECH-ADV-09).

Furthermore, the pair creation process fails to ensure whether the liquidation incentive BPS is within safe bounds, allowing overly high incentives (OS-ECH-ADV-05), and it is possible to font-run the creation of a new lending pair and market by registering an Aptos account at the pair's or the market's expected address (OS-ECH-ADV-03).

We also made recommendations for implementing proper validations (OS-ECH-SUG-03) and explicit checks to improve overall error handling (OS-ECH-SUG-04). We further suggested updating the code-base for improved functionality (OS-ECH-SUG-02) and efficiency (OS-ECH-SUG-01). Lastly, we advised adhering to coding best practices (OS-ECH-SUG-05) and removing redundant or unutilized code instances (OS-ECH-SUG-06).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/EchelonMarket/echelon-modules. This audit was performed against commit d5b0dd2.

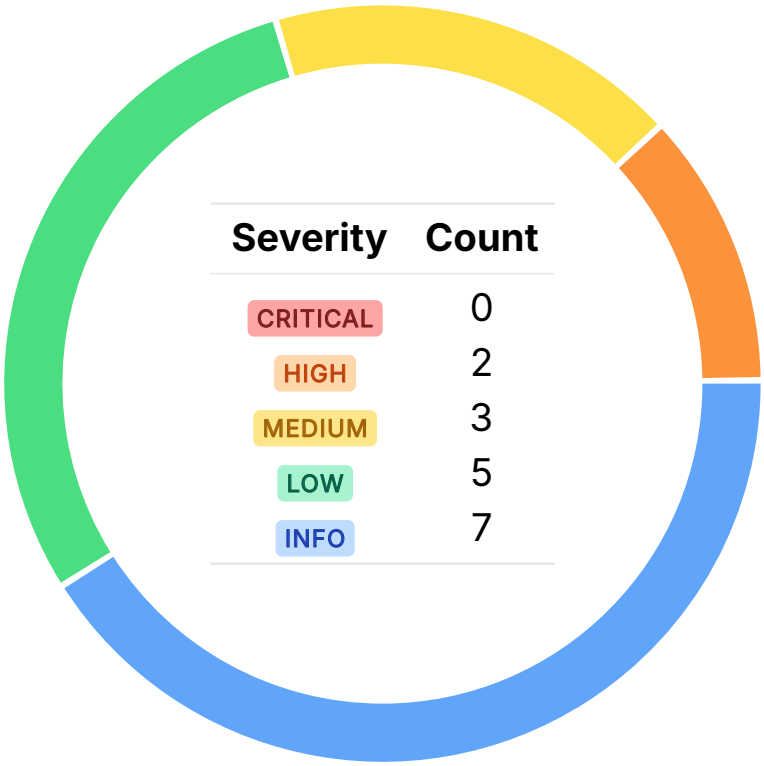
A brief description of the programs is as follows:

Name	Description
echelon-modules	It implements isolated lending markets, enabling users supply collateral, borrow assets, and earn interest within independent asset pairs.

03 — Findings

Overall, we reported 17 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	2
MEDIUM	3
LOW	5
INFO	7

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-ECH-ADV-00	HIGH	RESOLVED	A user's `last_acc_rewards_per_share` is initialized to the current `pool_acc_reward_per_share`, preventing users with an existing stake from accruing rewards retroactively when a new reward is added. This delays their reward accumulation until `accrue_user_pool_reward` is first called for them.
OS-ECH-ADV-01	HIGH	RESOLVED	`withdraw_internal` allows users to withdraw supply shares even when their position has bad debt, risking protocol insolvency.
OS-ECH-ADV-02	MEDIUM	RESOLVED	The current implementation allows unauthorized creation of a primary store at the package address or for the pool's asset, which may prevent the addition of farming rewards and block pool creation by making a valid primary store unavailable.
OS-ECH-ADV-03	MEDIUM	RESOLVED	It is possible to front-run the creation of a new lending pair and market by registering an Aptos account at the pair's or the market's expected address.
OS-ECH-ADV-04	MEDIUM	RESOLVED	`loop_supply_x_borrow_y_fa` incorrectly validates the swap route by asserting that the first asset matches the input and the last matches the output.

OS-ECH-ADV-05	LOW	RESOLVED	The pair creation process lacks validation of critical risk parameter. Also, there is a lack of boundary checks on liquidation parameters during market creation, risking invalid market configurations that may disrupt liquidation behavior.
OS-ECH-ADV-06	LOW	RESOLVED	`DEFAULT_ORIGINATION_FEE_BPS` value (10) is mislabeled as 0.01% in the comment, though it actually represents 0.10%.
OS-ECH-ADV-07	LOW	RESOLVED	`market_enter_efficiency_mode` fails to check if a market's liquidation incentive is higher than the efficiency mode's, allowing markets with weaker liquidation incentives to enter.
OS-ECH-ADV-08	LOW	RESOLVED	`claim_reward_fa` incorrectly utilizes `fungible_asset::withdraw`, preventing reward claims if the reward FA is dispatchable.
OS-ECH-ADV-09	LOW	RESOLVED	There is a lack of functionality to unfreeze coin stores after a freeze operation in `echelon_coin`.

Incorrect Reward Initialization HIGH OS-ECH-ADV-00

Description

The issue in the lending core farming module occurs when a user is accruing a specific reward for the first time. When a user first starts accruing a reward, their `last_acc_rewards_per_share` is set to `pool_acc_reward_per_share`. This is problematic, as users who had staked before the reward was introduced will not receive any rewards for the period between their staking and their first `accrue_user_pool_reward` call.

>_ *lending_core/sources/farming.move* RUST

```
fun accrue_user_pool_reward(pool: &Pool, user_pool: &mut UserPool, reward_name: String) {
    let pool_reward = simple_map::borrow(&pool.rewards, &reward_name);
    let pool_acc_reward_per_share = pool_reward.acc_rewards_per_share;
    if (!simple_map::contains_key(&user_pool.rewards, &reward_name)) {
        simple_map::upsert(&mut user_pool.rewards, reward_name, UserPoolReward {
            reward_name,
            last_acc_rewards_per_share: pool_acc_reward_per_share,
            reward_amount: 0,
        });
    };
    [...]
}
```

They will start receiving rewards only after `accrue_user_pool_reward` is first called for their address. This results in lost rewards for users who had an existing stake balance before the reward program was introduced. A similar issue also exists in the isolated lending farming module.

Remediation

Set `last_acc_rewards_per_share` to zero on first-time reward accrual to ensure users receive their full entitled rewards.

Patch

Fixed in a8c9301.

Missing Solvency Check HIGH OS-ECH-ADV-01

Description

`isolated_lending::withdraw_internal` lacks a check for bad debt, allowing users to withdraw supplied assets even if their position is underwater. This creates a vulnerability where users may extract value even though they are insolvent. If the borrowed value exceeds collateral, supply shares should not be withdrawable, as they may be needed to cover the shortfall.

Remediation

Enforce a solvency check before permitting withdrawals.

Patch

Fixed in 15a9dd7.

DOS Due to Blocking of Primary Store Creation MEDIUM OS-ECH-ADV-02

Description

The vulnerability relates to the creation of a primary fungible asset store in `new_reward_fa` in the lending core (shown below) and isolated core farming modules. `new_reward_fa` tries to create a primary store for the asset at the package address utilizing `create_primary_store`, which does not check if a store already exists at the address before creating a new one.

>_ *lending_core/sources/farming.move* RUST

```
/// Add a new Reward Fungible Asset for Farming.
public entry fun new_reward_fa(manager: &signer, asset_metadata: Object<Metadata>) acquires
    ↪  Farming {
    assert!(manager::is_manager(manager), ERR_FARMING_UNAUTHORIZED);
    primary_fungible_store::create_primary_store(package::package_address(),asset_metadata);
    new_reward_for_farming_internal(fungible_asset::name(asset_metadata));
}
```

Thus, `primary_fungible_store::create_primary_store` aborts if a primary store already exists at the address. As anyone may create a primary store at any address since it is permissionless, it enables an attacker to create a primary store for an asset at the package address. This action will block any subsequent attempts to add that address as a farming reward in both `lending_core` and `isolated_lending` farms.

>_ *lending_core/sources/lending.move* RUST

```
public fun create_market_with_jump_model_fa_v2([...]): Object<Market> acquires Lending,
    ↪  FungibleAssetInfo, Market, Vault, JumpInterestRateModel, PauseFlag, MarketRateLimit {
    [...]
    let market_addr = signer::address_of(&market_signer);
    primary_fungible_store::create_primary_store(market_addr, asset_metadata);
    move_to(&market_signer, FungibleAssetInfo{
        metadata: asset_metadata
    });
    [...]
}
```

Similarly, in the lending module, `create_market_with_jump_model_fa_v2` initializes a new lending market and calls `primary_fungible_store::create_primary_store`. This renders it vulnerable to frontrunning due to the predictability of the `market_addr`. An attacker monitoring the mempool will have sufficient information to derive the address and preemptively create a primary store for the pool's asset at the expected address, which will result in a `create_primary_store` call to fail when the original transaction executes.

Remediation

Utilize `ensure_primary_store_exists` instead of `create_primary_store`. This function will not attempt to create a new store if one already exists at the address.

Patch

Fixed in a8c9301.

Front-Running Pair/Market Creation MEDIUM OS-ECH-ADV-03

Description

`create_pair_with_jump_model` in `isolated_lending` is responsible for creating a new lending pair with a jump interest rate model. However, it is vulnerable to front-running. An attacker may observe a pending pair creation and preemptively register their Aptos account to the to-be-created pair's address before the pair is fully initialized.

>_ *isolated_lending/sources/isolated_lending.move* RUST

```
    public fun create_pair_with_jump_model<LiabilityCoinType, CollateralCoinType>(manager:
        ↪  &signer, collateral_factor_bps: u64, liquidation_incentive_bps: u64,
        ↪  initial_liquidity: u64, base_rate_bps: u64, multiplier_bps: u64,
        ↪  jump_multiplier_bps: u64, utilization_kink_bps: u64, collateral_dust_amount: u64):
        ↪  Object<Pair> acquires IsolatedLending, Pair, AccountPositions, CoinInfo,
        ↪  JumpInterestRateModel {
        [...]
        let (pair_signer, pair_obj) = create_pair_with_jump_model_internal(manager,
            ↪  collateral_factor_bps, liquidation_incentive_bps, initial_liquidity,
            ↪  liability_info, collateral_info, base_rate_bps, multiplier_bps,
            ↪  jump_multiplier_bps, utilization_kink_bps, collateral_dust_amount);

        aptos_account::create_account(signer::address_of(&pair_signer));
        [...]
}
```

Similarly, in `lending_core`, `create_market_with_jump_model_v2`, the market creation may be front-run with a call to register the Aptos account at the address of the market that it is going to be created, resulting in a denial-of-service scenario.

Remediation

`create_pair_with_jump_model` and `create_market_with_jump_model_v2` should create the Aptos account for the pair and market signer, respectively, only if it does not exist.

Patch

Fixed in a966e7f.

Inconsistency in Swap Route Validation MEDIUM OS-ECH-ADV-04

Description

`lending_leverage::loop_supply_x_borrow_y_fa`, there is a `pool_route` and an associated `asset_out_route`. These define how the borrowed asset `Y` is swapped back to the supplied asset `X` via a multi-hop route. `loop_supply_x_borrow_y_fa` contains incorrect assertions that check the start of the swap route against the input token and the end against an un-utilized `out_metadata` parameter. This logic is reversed and unnecessary. The route should instead start with the borrowed token (`Y`) and end with the input token (`X`), since the goal is to loop borrowed assets back into the original collateral.

>_ lending_leverage/sources/leverage.move RUST

```
public fun loop_supply_x_borrow_y_fa(account: &signer, supply_market: Object<Market>,
    ↪  borrow_market: Object<Market>, pool_route: vector<Object<Pool>>, asset_out_route:
    ↪  vector<Object<Metadata>>, target_health_factor_bps: u64, num_loop: u64, in:
    ↪  FungibleAsset, out_metadata: Object<Metadata>): FungibleAsset {
    assert!(vector::length(&pool_route) == vector::length(&asset_out_route),
        ↪  ERR_LEVERAGE_MALFORMED_ROUTE);
    assert!(*vector::borrow(&asset_out_route, vector::length(&asset_out_route) - 1) ==
        ↪  out_metadata, ERR_LEVERAGE_MALFORMED_ROUTE);
    assert!(*vector::borrow(&asset_out_route, 0) == fungible_asset::asset_metadata(&in),
        ↪  ERR_LEVERAGE_MALFORMED_ROUTE);
    [...]
}
```

Remediation

Remove the assert statement that checks whether the last token equals `out_metadata`, as it is unnecessary. Additionally, update the second assert to verify that the last asset in `asset_out_route` matches the input token, rather than the first.

Patch

Fixed in 26996e5.

Unchecked Liquidation Parameters LOW OS-ECH-ADV-05

Description

`isolated_lending::create_pair_internal` lacks sufficient validation of critical risk parameters when initializing a new lending pair. Specifically, the value of `liquidation_incentive_bps` is not bounded. `liquidation_incentive_bps` defines how much extra collateral a liquidator gets when they liquidate a position. A reasonable incentive encourages liquidators to help maintain protocol health. Since this is not verified against `MAX_LIQUIDATION_INCENTIVE_BPS`, it will be possible to set excessively high liquidation rewards. This may result in unfair liquidations where liquidators extract much more value than necessary. Similarly, there is no check to ensure that `liquidation_incentive_bps` is at least `BPS_BASE`.

>_ *isolated_lending/sources/isolated_lending.move* RUST

```
fun create_pair_internal(manager: &signer, collateral_factor_bps: u64,
    ↪  liquidation_incentive_bps: u64, initial_liquidity: u64, interest_rate_model_type: u64,
    ↪  collateral_dust_amount: u64, liability_info: AssetInfo, collateral_info: AssetInfo):
    ↪  (signer, Object<Pair>) acquires IsolatedLending, AccountPositions {
    // validate collateral_factor_bps
    assert!(collateral_factor_bps <= BPS_BASE,
        ↪  ERR_ISOLATED_LENDING_INVALID_COLLATERAL_FACTOR_BPS);
    validate_liquidation_collateral_coverage(collateral_factor_bps, liquidation_incentive_bps);

    // check initial_liquidity
    assert!(initial_liquidity >= MINIMUM_INITIAL_LIQUIDITY,
        ↪  ERR_ISOLATED_LENDING_INVALID_INITIAL_LIQUIDITY);

    // create pair
    let constructor_ref = object::create_sticky_object(package::package_address());
    let pair_signer = object::generate_signer(&constructor_ref);
    [...]
}
```

The pair creation logic further fails to check that `collateral_factor_bps` is strictly greater than zero, as a collateral factor of zero implies no borrowing power, which may render the pair useless. Also, `lending_core::create_market_with_jump_model_internal` lacks sufficient boundary checks on key parameters. Specifically, it does not ensure that `liquidation_threshold_bps` is less than or equal to 10,000 (100%) or that `liquidation_incentive_bps` is greater than or equal to 10,000.

Remediation

Add an assertion to prevent `liquidation_incentive_bps` from exceeding `MAX_LIQUIDATION_INCENTIVE_BPS` and from going below `BPS_BASE` . Also, verify that `collateral_factor_bps` to be greater than zero, and ensure `liquidation_threshold_bps <= BPS_BASE` and `liquidation_incentive_bps >= BPS_BASE` in `create_market_with_jump_model_internal`.

Patch

Fixed in a6bec7e.

Irregularity in Fee Comment Annotation LOW OS-ECH-ADV-06

Description

The `DEFAULT_ORIGINATION_FEE_BPS` constant in `lending_core` is set to 10 with a comment stating it represents 0.01%, which is incorrect as 10 basis points equals 0.10%, not 0.01%. This mismatch creates ambiguity regarding the actual fee applied on borrow transactions. If the intended fee is 0.01%, the value should be 1, and if the value of 10 is correct, the comment should be updated to state 0.10%.

>_ *lending_core/sources/lending.move* RUST

```rust
const DEFAULT_ORIGINATION_FEE_BPS: u64 = 10; // 10 bps or 0.01% of the borrowed asset
```

Remediation

Ensure the comment and the assigned value are consistent with each other.

Patch

Fixed in a966e7f.

Insufficient Liquidation Incentive Check LOW OS-ECH-ADV-07

Description

In `lending_core`, `create_efficiency_mode_v2` requires that any market added to an `eMode` have a liquidation incentive greater than the `eMode`'s, ensuring strong liquidation guarantees. However, `market_enter_efficiency_mode` does not enforce this check, allowing a manager to later add markets with lower liquidation incentives. This bypasses the intended safety constraint, enabling borrowers to gain excessive leverage while reducing rewards for liquidators.

Remediation

Add the liquidation incentive check in `market_enter_efficiency_mode` similar to what is implemented in `create_efficiency_mode_v2`.

Patch

Fixed in a966e7f.

Flaw in Reward Withdrawal Logic LOW OS-ECH-ADV-08

Description

In `lending::farming` (shown below), `claim_reward_fa` withdraws rewards utilizing `fungible_asset::withdraw`. However, if the reward token is a dispatchable fungible asset, this withdrawal will fail, preventing users from claiming their rewards. A similar issue exists within `claim_reward_fa` in `isolated_lending`.

>_ lending_core/sources/lending.move RUST

```
public fun claim_reward_fa(account: &signer, asset_metadata: Object<Metadata>,
    ↪  farming_identifier: String): FungibleAsset acquires Farming, Staker {
    let reward_amount = claim_reward_internal(signer::address_of(account),
        ↪  fungible_asset::name(asset_metadata), farming_identifier);
    fungible_asset::withdraw(&package::package_signer(),
        ↪  primary_fungible_store::primary_store(package::package_address(), asset_metadata),
        ↪  reward_amount)
}
```

Remediation

Modify `claim_reward_fa` to utilize `primary_fungible_store::withdraw` to dynamically handle both non-dispatchable and dispatchable assets.

Patch

Fixed in a6bec7e and 809241d.

Lack of Unfreeze Functionality LOW OS-ECH-ADV-09

Description

`echelon_coin` allows freezing stores but lacks a corresponding function to unfreeze them, resulting in permanent lockups. Add a function to unfreeze the store's proper access control to ensure proper functionality.

Remediation

Implementing a corresponding unfreeze functionality to revert a freeze operation.

Patch

Fixed in a966e7f.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-ECH-SUG-00	Several liquidation functions mistakenly borrow the wrong asset due to incorrect flashloan parameter ordering, resulting in them executing with zero funds and failing.
OS-ECH-SUG-01	The codebase may be optimized for improved efficiency.
OS-ECH-SUG-02	Recommendation for updating the codebase to improve functionality and mitigate potential security issues.
OS-ECH-SUG-03	There are several instances where proper validation is not performed, resulting in potential security issues.
OS-ECH-SUG-04	Modifications to ensure the inclusion of explicit checks to prevent unexpected aborts or panics, improving the protocol's robustness and error handling.
OS-ECH-SUG-05	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-ECH-SUG-06	The codebase contains multiple cases of redundant and unutilized code that should be removed for better maintainability and clarity.

Failed Liquidations Due to Flashloan Asset Mismatch OS-ECH-SUG-00

Description

In the current implementation, several functions in `liquidator` incorrectly order parameters in `flashloan` calls, resulting in failed liquidations due to borrowing the wrong asset. For instance, in `repay_APT_seize_THL`, the `repay_amount` (intended to repay `APT` debt) is mistakenly passed as the first parameter, resulting in `THL` to be borrowed instead. `THL` is actually borrowed and stored in the `zero_0` variable. Consequently, the correct asset is not received, the `loan` variable remains empty, and the liquidation proceeds with zero assets, failing with no debt repaid or collateral seized.

liquidator/sources/liquidator.move RUST

```
public entry fun repay_APT_seize_THL(
    account: &signer,
    vault_address: address,
    repay_amount: u64,
    min_shares_out: u64
) {
    let (zero_0, loan, zero_2, zero_3, flashloan) = stable_pool::flashloan<ThalaAPT, AptosCoin,
        ↪  Null, Null>(
        repay_amount,
        0,
        0,
        0
    );
    let seized = liquidate<THL, AptosCoin>(account, vault_address, loan, min_shares_out);
    let swapped = weighted_pool::swap_exact_in<THL, AptosCoin, Null, Null, Weight_50, Weight_50,
        ↪  Null,Null, THL, AptosCoin>(seized);
    repay_amount = repay_amount + math64::mul_div(repay_amount, 1, 10000);
    stable_pool::pay_flashloan(zero_0, coin::extract(&mut swapped, repay_amount), zero_2,
        ↪  zero_3,flashloan);
    aptos_account::deposit_coins<AptosCoin>(signer::address_of(account), swapped);
}
```

Remediation

Ensure all functions in `liquidator` borrow the correct assets for liquidation by using the proper parameter order in `flashloan` calls.

Patch

Acknowledged by Thala team.

Code Optimization OS-ECH-SUG-01

Description

1. In `lending`, within `create_efficiency_mode_v2`, the assertion that checks `collateral_factor_bps <= liquidation_threshold_bps` is placed inside the loop even though these values are constant across the loop. Similarly, in `set_efficiency_mode_liquidation_threshold_bps`, the `e_mode.collateral_factor_bps <= liquidation_threshold_bps` assertion is placed inside the loop. Repeating these assertions for every market is unnecessary. This results in inefficiency due to redundant computations. Move these two assertions outside their respective loops.

>_ lending_core/sources/lending.move RUST

```
public entry fun create_efficiency_mode_v2(manager: &signer, markets:
    ↪  vector<Object<Market>>, collateral_factor_bps: u64, liquidation_incentive_bps: u64,
    ↪  liquidation_threshold_bps: u64) acquires Lending, Market,
    ↪  MarketLiquidationThreshold, MarketLiquidationIncentive, EmodeLiquidationThreshold
    ↪  {
    [...]
    vector::for_each(emode_markets, |market_obj| {
        [...]
        assert!(collateral_factor_bps <=
            ↪  liquidation_threshold_bps,ERR_LENDING_INVALID_LIQUIDATION_THRESHOLD_BPS);
        [...]
    });
}
```

2. In `lending_core`, `set_market_liquidation_threshold_bps`, `set_market_rate_limit_internal`, and `set_market_liquidation_paused` currently utilize manual signer generation, which duplicates logic. Replacing these with the standardized `market_signer` inline function will be more optimal.
3. `init_module` in `echelon_coin` unnecessarily mints and burns coins and inefficiently handles capabilities by borrowing them after moving. It may be simplified by utilizing `zero_coin` and utilizing the `Capabilities` before moving them.
4. `lending_core::withdraw_reserve` manually handles coin withdrawal, registration, and deposit, which introduces unnecessary complexity. Replace this with a call to `aptos_account::transfer_coins` to reduce boilerplate and improve readability.

>_ *lending_core/sources/lending.move* RUST

```rust
// Reserve is owned by the protocol. The funds can be withdrawn as protocol income.
public entry fun withdraw_reserve<CoinType>(manager: &signer, market_obj: Object<Market>,
    ↪  recipient: address) acquires Market, CoinInfo {
    [...]
    market.total_cash = market.total_cash - withdraw_amount;
    let _total_cash = market.total_cash;

    let withdrawn_coins = coin::withdraw<CoinType>(&market_signer(market_obj),
        ↪  withdraw_amount);
    if (signer::address_of(manager) == recipient) {
        coin::register<CoinType>(manager);
    };
    coin::deposit(recipient, withdrawn_coins);
    [...]
}
```

Remediation

Incorporate the above-stated refactors.

Code Refactoring OS-ECH-SUG-02

Description

1. In `isolated_lending` there are `#[view]` functions that are read-only. The `#[view]` attribute is intended for read-only functions that do not mutate blockchain state. However, `account_supply_amount` is incorrectly marked as a view function, even though it calls `accrue_pair_interest`, which updates the `Pair` object's internal state. Remove the `#[view]` decorator to avoid confusion.

   >_ *isolated_lending/sources/isolated_lending.move* RUST

   ```
   #[view]
   public fun account_supply_amount(account_addr: address, pair_obj: Object<Pair>): u64
       ↪  acquires IsolatedLending, AccountPositions, Pair, JumpInterestRateModel {
       let shares = account_supply_shares(account_addr, pair_obj);
       if (shares == 0) return 0;

       accrue_pair_interest(pair_obj);
       supply_shares_to_amount(pair_obj, shares)
   }
   ```

2. In the current implementation of the `liquidator` module, the flashloan fee is statically set to the default value of 1 basis point (`0.01%`). However, this may not accurately reflect the actual fee charged by the flashloan provider, especially if the fee structure changes or differs across assets or platforms. To improve accuracy and adaptability, the repay functions should invoke `flashloan_fee_bps` to dynamically retrieve the precise fee applicable to the asset borrowed.
3. Entry function wrappers that wrap a deprecated functionality, such as `create_market_with_jump_model_fa` in `lending_core::scripts` and `create_efficiency_mode` in `lending_core`, should also be marked as deprecated to prevent the utilization of non-functional logic.
4. Currently, in `farming`, relying on the name of a `Coin` or `FungibleAsset` to identify rewards is unsafe, as names are not unique and may be duplicated across different types. Ensure the reward is clearly distinguishable. Otherwise, it may result in issues i.e., where new epochs are mistakenly initiated for different fungible assets that share the same name, potentially resulting in incorrect transfers.

Remediation

Incorporate the above-stated refactors.

Missing Validation Logic

OS-ECH-SUG-03

Description

1. There is an inconsistency in how `collateral_factor_bps` is validated across `lending_core`. In `set_market_collateral_factor_bps`, it is allowed to be equal to the liquidation threshold (`<=`), while in `create_market_with_jump_model_internal`, it must be strictly less than (`<`). This mismatch may allow risky configurations with no liquidation buffer, increasing the chances of bad debt. The validation logic should be unified across all relevant functions to consistently enforce `collateral_factor_bps < liquidation_threshold_bps`, including efficiency mode setters.

>_ *lending_core/sources/lending.move* RUST

```
public entry fun set_market_collateral_factor_bps([...]) acquires Market, Lending,
    ↪  MarketLiquidationThreshold {
    [...]
    assert!(collateral_factor_bps <=
        ↪  borrow_market_liquidation_threshold(market_obj).value_bps,
        ↪  ERR_LENDING_INVALID_LIQUIDATION_THRESHOLD_BPS);
    [...]
}

fun create_market_with_jump_model_internal([...]): (signer, Object<Market>) acquires
    ↪  Lending, MarketRateLimit, Market, MarketOriginationFee {
    assert!(utilization_kink_bps <= BPS_BASE, ERR_LENDING_INVALID_UTILIZATION_KINK_BPS);
    assert!(collateral_factor_bps < liquidation_threshold_bps,
        ↪  ERR_LENDING_INVALID_COLLATERAL_FACTOR_BPS);
    [...]
}
```

2. `lending_core::create_market_internal` currently permits `collateral_factor_bps` to be set equal to `BPS_BASE`, which is problematic because it eliminates the required liquidation buffer. Specifically, since the liquidation threshold is also capped at `BPS_BASE`. Enforce `collateral_factor_bps < BPS_BASE`. Additionally, it will be prudent for collateral factor setters to verify if `collateral_factor_bps > 0`.

>_ *lending_core/sources/lending.move* RUST

```
fun create_market_internal([...]): (signer, Object<Market>) acquires Lending {
    // validate collateral_factor_bps
    let lending = borrow_global_mut<Lending>(package::package_address());
    assert!(collateral_factor_bps <= BPS_BASE, ERR_LENDING_INVALID_COLLATERAL_FACTOR_BPS);
    [...]
}
```

3. In `isloated_lending`, allowing a `Pair` with the same `Coin` or `FungibleAsset` for both liability and collateral undermines the collateralization logic, enabling users to borrow against their own deposits without real risk. An assertion should enforce asset-type distinction during pair creation.
4. The protocol should extend slippage protection to functions beyond liquidation and include checks to ensure returned shares or amounts are not zero. It should also validate that input amounts passed to functions are non-zero.

Remediation

Include the above-mentioned validations.

Error Handling

OS-ECH-SUG-04

Description

1. To ensure smooth operation and graceful handling of failures, view-only functions should verify the existence of a resource before borrowing it. For example, `farming::total_alloc_point` should check whether the specified `reward_name` exists, as it currently assumes its presence in the table, which may result in an abort if it is missing.
2. Multiple function within `lending_core` access a user's `Vault` without first checking if it actually exists. This may result in unclear aborts. Add a `vault_exists(account_addr)` check before borrowing to prevent unexpected failures. Similarly, `lending_core::paused` should check if the `pause` flag exists before borrowing to improve error handling, as it may not be set at all.
3. `lending_core::scripts` directly withdraws coins or fungible assets without checking if the account has sufficient balance, which may result in abrupt transaction failures. Add checks for balance to enable more graceful error handling and clearer failure messages.
4. `burn_fa` in `echelon_coin` should verify whether the incoming fungible asset's metadata matches the `burn_ref`. This improves error handling.

>_ echelon_coin/sources/echelon_coin.move RUST

```
public fun burn_fa(manager: &signer, fa: FungibleAsset) acquires Capabilities {
    assert!(manager::is_manager(manager), ERR_ECHELON_COIN_UNAUTHORIZED);

    let cap = borrow_global<Capabilities>(package::package_address());
    let (burn_ref, burn_ref_receipt) = coin::get_paired_burn_ref(&cap.burn_capability);
    fungible_asset::burn(&burn_ref, fa);
    coin::return_paired_burn_ref(burn_ref, burn_ref_receipt);
}
```

5. `lending_core::deposit_reserve_fa` assumes the target market supports fungible assets, but currently lacks a check to enforce this. Adding a validation to confirm the market type will improve error handling on encountering incompatible market types.
6. In `isolated_lending`, relevant functions should register the `CoinType` to the recipient before performing a transfer to prevent avoidable errors. For example, `liquidate` should register the `CollateralCoinType` to the liquidator, and `borrow_entry` should register the borrowed `CoinType` to the borrower's account.

Remediation

Update the codebase with explicit checks to ensure proper error handling.

Code Maturity OS-ECH-SUG-05

Description

1. In `lending`, most setter functions begin with a call to `assert_market_exists(market_obj)`. This ensures the `market_obj` passed to the function refers to a valid, initialized market before proceeding. However, `set_market_liquidation_incentive_bps` and `set_market_rate_limit_internal` do not call `assert_market_exists`. Ensure all setter functions verify that the `market_obj` exists. Similarly, in `isolated_lending`, `set_pair_liquidation_incentive_bps` should call `assert_pair_exists` to confirm that such a `Pair` exists, as other pair setters do.

>_ isolated_lending/sources/isolated_lending.move RUST

```
public entry fun set_pair_liquidation_incentive_bps(manager: &signer, pair_obj:
    ↪  Object<Pair>, liquidation_incentive_bps: u64) acquires Pair {
    assert!(manager::is_manager(manager), ERR_ISOLATED_LENDING_UNAUTHORIZED);
    assert!(liquidation_incentive_bps >= BPS_BASE && liquidation_incentive_bps <=
        ↪  MAX_LIQUIDATION_INCENTIVE_BPS,
        ↪  ERR_ISOLATED_LENDING_INVALID_LIQUIDATION_INCENTIVE_BPS);

    let pair = borrow_mut_pair(pair_obj);
    validate_liquidation_collateral_coverage(pair.collateral_factor_bps,
        ↪  liquidation_incentive_bps);

    // update liquidation_incentive_bps
    pair.liquidation_incentive_bps = liquidation_incentive_bps;
}
```

2. State-changing functions should emit events to provide transparency and allow off-chain systems to track protocol changes. Emitting events for important updates, such as changing interest fees, setting `pause` flags, or updating market/lending parameters, helps with debugging and improves user awareness on the current protocol state. Events should only be emitted when an actual change occurs to avoid misleading logs and unnecessary noise.

>_ lending_core/sources/lending.move RUST

```
public entry fun set_paused(manager: &signer, paused: bool) acquires PauseFlag {
    assert!(manager::is_manager(manager), ERR_LENDING_UNAUTHORIZED);
    assert!(pause_flag_exists(), ERR_LENDING_PAUSE_FLAG_UNINITIALIZED);
    let flag = borrow_global_mut<PauseFlag>(package::package_address());
    flag.paused = paused
}
```

3. In `echelon_coin` `package` module should verify that the `Package` does not already exist in `init_module` and implement an `initialized` function to enforce this check in `package_signer`. This improves consistency and aligns with the pattern utilized in other package modules in the project.

Remediation

Implement the above-mentioned suggestions.

Unutilized/Redundant Code

OS-ECH-SUG-06

Description

1. In `isolated_lending`, some internal preview functions, such as `preview_remove_collateral`, are incorrectly annotated with `#[view]` despite mutating protocol state. Although currently unutilized, these functions pose a significant risk if called, as they may perform critical actions like wiping a user's collateral without validation or fund transfers. Such functions should be removed.

>_ isolated_lending/sources/isolated_lending.move RUST

```
#[view]
fun preview_remove_collateral(account_addr: address, pair_obj: Object<Pair>, amount: u64):
    ↪  (FixedPoint64, FixedPoint64) acquires AccountPositions, Pair, CoinInfo,
    ↪  FungibleAssetInfo, IsolatedLending, JumpInterestRateModel {
    remove_collateral_internal(account_addr, pair_obj, amount, false);
    account_liquidity(account_addr, pair_obj)
}
```

2. In `farming::init_alloc_point`, the final `if/else` block setting `last_rewards_sec` is redundant, as the presence of the `reward_name` in `farming.rewards` is already ensured by a prior assertion.

>_ lending_core/sources/farming.move RUST

```
/// Add a new Reward for a specific farming pool.
public entry fun init_alloc_point(manager: &signer, reward_name: String,
    ↪  farming_identifier: String, alloc_point:u64) acquires Farming {
    [...]
    // check Reward has been initialized for Farming
    assert!(simple_map::contains_key(&farming.rewards, &reward_name),
        ↪  ERR_FARMING_REWARD_UNINITIALIZED);
    [...]
    // update Pool
    let pool = simple_map::borrow_mut(&mut farming.pools, &farming_identifier);
    let last_rewards_sec = if (simple_map::contains_key(&farming.rewards, &reward_name)) {
        current_epoch_seconds(simple_map::borrow(&farming.rewards, &reward_name))
    } else {
        0
    };
    [...]
}
```

3. There are multiple unutilized constant declarations (such as `ERR_PACKAGE_UNAUTHORIZED` in `lending_maganer::package`), which should be removed for improved readability.
4. There are some private, deprecated `preview_*` functions in `lending_core` that are not utilized. They should be removed.
5. In the current implementation, repeated borrows in several places across the code introduce avoidable overhead. For reference, in `lending::accrue_market_interest`, `market` is borrowed three times instead of just once. Similarly, `supply_internal` in `isolated_lending` borrows `Pair` three times. These borrows may be consolidated in each function to avoid unnecessary storage reads and associated gas costs. Also, view functions borrow mutably when it is not necessary (`farming::pool_stake_amount` borrows `Farming` and `Pool` mutably).
6. `lending_core::set_market_liquidation_threshold_bps` unnecessarily calls `validate_liquidation_threshold_and_incentive` twice. Also, `lending_core::is_shortfall` is unutilized and may be removed. Furthermore, `market_asset_metadata` call in `lending_core::create_market_with_jump_model_fa_v2` is unnecessary and may be removed.

Remediation

Remove the redundant and unutilized code instances highlighted above.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.