Aave Aptos V3.0.2 Periphery Security Review

Auditors

Christoph Michel, Lead Security Researcher

Emanuele Ricci, Lead Security Researcher

T1moh, Associate Security Researcher

Jay, Security Researcher

Report prepared by: Lucas Goiriz

June 18, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Aave Labs creates smart contract-enabled products and public goods (open source protocols) that incorporate decentralized blockchain technologies and token-based economies.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Aave Aptos V3.0.2 Periphery according to the specific commit. Any modifications to the code will require a new security review.

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

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 38 days in total, Aave Labs engaged with Spearbit to review the aptos-v3-sb-audit protocol. In this period of time a total of **24** issues were found.

Summary

Project Name	Aave Aptos V3.0.2 Periphery
Repository	aptos-v3-sb-audit
Commit	3668ac01
Type of Project	DeFi, Lending
Audit Timeline	Mar 12th to Apr 19th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	3	3	0
Low Risk	11	9	2
Gas Optimizations	0	0	0
Informational	10	10	0
Total	**24**	**22**	**2**

5 Findings

5.1 Medium Risk

5.1.1 Tokens cannot be withdrawn from `admin_controlled_ecosystem_reserve.move`

Severity: Medium Risk

Context: admin_controlled_ecosystem_reserve.move#L113-L115

Description: Module `admin_controlled_ecosystem_reserve.move` is supposed to receive incentive tokens which will act as reward in `reward_controller.move`, specifically it will receive APT.

It maps `token` → `FungibleStore`:

```
struct AdminControlledEcosystemReserveData has key {
    fungible_assets: SmartTable<Object<Metadata>, Object<FungibleStore>>, // <<<
    extend_ref: ObjExtendRef,
    transfer_ref: ObjectTransferRef,
    funds_admin: address
}
```

The problem is that `fungible_assets` is never written, so the function `transfer_out()` does nothing:

```
public fun transfer_out(
    sender: &signer,
    asset_metadata: Object<Metadata>,
    receiver: address,
    amount: u64
) acquires AdminControlledEcosystemReserveData {
    check_is_funds_admin(signer::address_of(sender));

    let admin_controlled_ecosystem_reserve_data =
        borrow_global_mut<AdminControlledEcosystemReserveData>(
            admin_controlled_ecosystem_reserve_address()
        );

    if (smart_table::contains(
        &admin_controlled_ecosystem_reserve_data.fungible_assets, asset_metadata
    )) {
        // ...
    }
}
```

There is no way to transfer out stored tokens, so they can't be used as incentives.

Recommendation: There are 2 ways:

1. Add a mirror function `transfer_in()` which writes to `fungible_assets`.
2. Refactor this module to store tokens in Resource Account, this way there is no need to have map `fungible_assets` because tokens are stored in primary fungible store. The flow is the following:
   - Incentive tokens are sent to Resource Account's primary fungible store.
   - This module contains `SignerCapability` of that account, so can transfer out those tokens.

Aave Labs: Fixed in PR 383.

Spearbit: Fix verified.

5.1.2 Incentives cannot be configured

Severity: Medium Risk

Context: emission_manager.move#L110

Description: `emission_manager::configure_assets()` is supposed to initialize reward parameters. It uses struct `RewardsConfigInput` as argument:

```
public fun configure_assets(
    account: &signer, config: vector<RewardsConfigInput>
) acquires EmissionManagerData {
```

However this struct is defined in module `rewards_controller.move`. In Move language struct can be created only in the same module where it's defined. `rewards_controller.move` does not contain functions to create `RewardsConfigInput`, therefore `emission_manager::configure_assets()` can't be executed. As a result, there is no way to call `emission_manager::configure_assets()` and therefore configure rewards.

Recommendation: Refactor functions to use raw values instead of combining them to struct `RewardsConfigInput`.

Aave Labs: Fixed in PR 406.

Spearbit: Fix verified.

5.1.3 `collector` and `AToken` treasury incompatibilities

Severity: Medium Risk

Context: collector.move#L1, collector.move#L153

Description: On Aave Solidity implementation, the `Collector` contract is the contract used as the `AToken` Treasury for every `AToken` deployed. They have deployed it once, and every `AToken` is configured with that instance. This means that all the `AToken` shares minted to the Treasury as fees on the interest/flashloan are minted to the address of that contract.

The current deployed address on Ethereum Mainnet is this: 0x464C71f6c2F760DdA6093dCB91C24c39e5d6e18c.

The specifications to have a working 1:1 version on Aptos are these:

1. The module must offer a single `address` to receive the `AToken` shares minted to it. Following the approach used in the `a_token_factory`, the contract should generate a **resource account**, save the `SignerCapability` returned by `account::create_resource_account` and save it to later on be able to move funds.
2. The module should offer a way to withdraw `AToken` shares from Aave Protocol. This function must be authorized.
3. The module should offer a way to transfer `AToken` shares to an `receiver`. This function must be authorized.
4. The module should offer a general-purpose function to transfer a `FungibleAsset/DispatchableFungibleAsset` token to a `receiver`. This function must be authorized.
5. The module should be able to claim rewards from the Aave Reward system. This function must be authorized.

The second and third features are optional, but at least one (or both) **must** be implemented. It is up to the Aave Protocol team to decide which would be the common use case scenario given the usage on the Aave Protocol EVM.

- Problem 1: There's no documentation relative to the `treasury` attribute of `a_token_factory::TokenData`: Aave Protocol must document it and explain what will be used as the `address` of the `treasury`.
- Problem 2: `collector` module is incompatible as an Aave Protocol reserve: Let's assume that the current implementation of `collector` is used as the `AToken` Treasury.
    1. The secondary storage generated for each `FungibleAsset` is **incompatible** with the `AToken` minting process.

       Let's assume that a secondary store for `aUSDC` is generated via `collector::generate_secondary_collector_fungible_store` and the `address` of that secondary store is used as the `aUSDC.treasury`. The process of minting the `aUSDC` shares for the Treasury will send those shares to a different wallet, to be

specific, to the one identified by the address `primary_fungible_store::ensure_primary_store_exists(on_behalf_of, asset)` where `on_behalf_of` is the `address` of the **secondary store** associated with `aUSDC` in the `collector` module.

Calling `collector::withdraw` to withdraw will fail because the secondary store has zero balance of those shares; the shares are owned by the **primary store** of the **secondary store**.

2. Depositing a fungible asset into the Treasury on purpose does not make sense. The Aave Treasury has the main purpose of receiving the fees generated by the Aave Protocol itself, not receiving general `FungibleAsset`. This function should be removed.
3. `collector` is `incompatible` with the `AToken`.

   When `AToken` shares are minted to or transferred to a wallet, that wallet will be **frozen**, and the only way to transfer those shares from the primary store associated with an `address` is to execute the "custom" function `aave_pool::pool_token_logic::transfer(sender: &signer, recipient: address, amount: u256, a_token_address: address)`.

   The `collector::withdraw` implementation is currently incompatible (on top of the other issues) with withdraw/transfer of the `AToken` shares received.

- Problem 3: the `collector` cannot claim rewards earned with the Aave Reward system: Even if the `treasury` does not "directly" supply underlying into Aave Protocol itself, it will anyway receive `AToken` shares that will make it automatically eligible for rewards in the Aave Reward system if the `AToken` asset has been configured with some distributions.

  Given that the `Collector` contract on Solidity has no way to interact with the `RewardsController` contract, the options are only 2:

  1. A `claimer` that can claim on behalf of the `treasury` has been configured.
  2. The `treasury` simply won't claim the rewards.

  Given that currently the `claimer` for the Treasury address is `address(0)` we can assume that the Treasury won't claim those rewards.

  Aave Protocol should anyway provide an explicit statement about this topic.

- Problem 4: Lack of tests:
  - The `collector_tests` test module lacks the tests that should validate the correct behavior and all the possible exceptions.
  - There are no tests that prove the whole flow: mint `AToken` Treasury shares to the `treasury` and withdraw/transfer them.

  Aave Protocol should implement all the needed tests cases to prove that.

  1. All the features have been correctly implemented for the `Collector`.
  2. `AToken` shares can be minted to the `Collector` module and then withdrawn from it.

In general, we need to stress out that we would expect to have at least (where possible) 1:1 parity between the test suite developed for the Solidity codebase.

Recommendation:

1. Aave Protocol should explain with detail how the `AToken` treasury address will be configured with and how they plan to withdraw/transfer the `AToken` shares minted to the treasury primary storage.
2. Aave Protocol should explain with an explicit statement if the `treasury` will claim the rewards accrued by the Treasury itself in the Aave Reward system and how they plan to claim them.
3. Aave Protocol should create all the tests needed for both the Aave Treasury module and the full flow from minting AToken shares to the Treasury to withdrawing them from the Treasury primary store.
4. If the `collector` module is the module used as the treasury, it must be fully refactored and should implement all the needed functions explained above. Otherwise, the module should be removed from the codebase.

Here are some practical examples to that can be seen as possible recommendations:

- The `collector` module during the `init_module` execution creates a single resource acciunt and stores it's `SignerCapability` into the `CollectorData` global storage.
- The address of the signer bound to the `SignerCapability` will be used as the `AToken` treasury when the reserve is initialized.
- This resource account will receive the `AToken` shares that are minted to the primary fungible store.
- When needed, the `collector::transfer` will re-generate the signer starting from the capability and use it to call `pool_token_logic::transfer` to transfer the `AToken` shares. The same logic can be applied if there's the need to withdraw the underlying by burning the `AToken` shares.
- The `CollectorData` struct does not need to store the `fungible_assets` mapping, the `transfer_ref` and `extend_ref` anymore, they can be removed.

Aave Labs: Fixed in PR 393.

Spearbit: Fix verified.

5.2 Low Risk

5.2.1 `coin_to_fa` should revert if the user has not enough `CoinType` balance to perform the conversion

Severity: Low Risk

Context: coin_migrator.move#L36-L39

Description: The current sanity check performed by `aave_pool::coin_migrator::coin_to_fa` on the user balance is not correctly implementing the requirement to revert when the caller has not enough `CoinType` balance to perform the conversion of `amount` coins.

The function fetches the user's balance by calling `coin::balance<CoinType>(signer::address_of(account))` which does not strictly return the amount of `CoinType` owned by the user that could be converted to the `FungibleAsset` version, but it rather returns the total amount of both `CoinType` + `FungibleAsset` (of the `CoinType`) amount.

```
**[view]:**

/// Returns the balance of `owner` for provided `CoinType` and its paired FA if exists.
public fun balance<CoinType>(owner: address): u64 acquires CoinConversionMap, CoinStore {
    let paired_metadata = paired_metadata<CoinType>();
    coin_balance<CoinType>(owner) + if (option::is_some(&paired_metadata)) {
        primary_fungible_store::balance(
            owner,
            option::extract(&mut paired_metadata)
        )
    } else { 0 }
}
```

As an example:

- Alice owns `100 CoinA`.
- Alice "convert" `10 CoinA` to `10 FA_CoinA`.

If you call `coin::balance<CoinType>(alice)` it will return `100` not `90`.

Recommendation: A possible solution to implement the needed requirements would be something similar to this:

```
fun get_fa_balance<CoinType>(account: address): u64 {
    let wrapped_fa_meta = coin::paired_metadata<CoinType>();

    if( option::is_some(&wrapped_fa_meta) ) {
        let wrapped_fa_meta = option::destroy_some(wrapped_fa_meta);
        let user_fa_store =
            primary_fungible_store::ensure_primary_store_exists(
                account, wrapped_fa_meta
            );

        fungible_asset::balance(user_fa_store)
    } else {
        return 0
    }
}

public entry fun coin_to_fa<CoinType>(account: &signer, amount: u64) {
    let complete_balance = coin::balance<CoinType>(signer::address_of(account));
    let fa_balance = get_fa_balance<CoinType>(signer::address_of(account));
    assert!(
        complete_balance - fa_balance >= amount,
        error_config::get_einsufficient_coins_to_wrap()
    );

    /// rest of the logic
}
```

In additional to the above check, Aave Protocol should also revert if the user requests to convert a number of coins equal to `0`. Internally, the Aptos framework implementation does allow it but on the Aave Protocol said it would emit a useless `CointToFaConvertion` event.

Aave Labs: Fixed in PR 309.

Spearbit: Fix verified.

5.2.2 Lack of lower and Upper bound in `set_emission_per_second`

Severity: Low Risk

Context: rewards_controller.move#L1114-L1118

Description: The `set_emission_per_second` function in the RewardsController module allows setting new emission rates for rewards without validating upper or lower bounds. This could lead to two potential issues:

1. Setting extremely high emission rates could cause excessive rewards distribution and potential numerical overflow when calculating rewards.
2. Setting extremely low (but non-zero) emission rates could lead to rewards that effectively round to zero, wasting gas on calculations that produce no meaningful rewards.

- Formal Verification Test Condtion:

```
let emission_per_second = (reward_data.emission_per_second as u256);
// ...

// This line shows the upper bound check
aborts_if emission_per_second > MAX_EMISSION_RATE;
ensures emission_per_second <= MAX_EMISSION_RATE;
```

- Failed Condition:

```
error: abort not covered by any of the `aborts_if` clauses
 rewards_controller.move:XXX
     aborts_if emission_per_second > MAX_EMISSION_RATE;
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
     The function does not abort under this condition
```

Recommendation: Consider adding upper and lower bound validation checks before setting the new emission rate. Here's is what a possible validation fix could look like:

```
assert!(
    new_emission_per_second <= MAX_EMISSION_RATE,
    error_config::get_eemission_rate_too_high()
);
assert!(
    new_emission_per_second == 0 || new_emission_per_second >= MIN_EMISSION_RATE,
    error_config::get_eemission_rate_too_low()
);
```

Aave Labs: Fixed in PR 408.

Spearbit: Fix verified.

5.2.3 Aave Reward System Management, documentation and concerns

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: On Aave Protocol EVM, the Reward system is currently configured as follows: each `reward` has its own `TransferStrategy` which will transfer the rewards (depending on the strategy itself) from a specific `REWARDS_VAULT`.

From the research done by Spearbit for Aave Reward system on the Ethereum Mainnet, every configured reward uses the same deployed `PullRewardsTransferStrategy` that will pull from the same `REWARDS_VAULT`. The `REWARDS_VAULT` configured is the "ACI multisig address" 0xac140648435d03f784879cd789130F22Ef588Fcd, which in practice can be considered a Smart Wallet.

We assume that, on Solidity, the `ACI multisig` will provide just the allowance needed by the `PullRewardsTransferStrategy` to correctly limit what an external source (like the strategy) can pull in case of a problem/hack.

- **Problem 1: lack of proper explanation and documentation relative to the `rewards_vault`:** On the Aptos side, we only know what has been provided as a "dev comment" on top of the `rewards_vault` attribute itself.

```
// An object representing the pull-reward transfer strategy
//
// This strategy pulls rewards (as `FungibleAsset`) from a vault resource
// account to the recipient address.
#[resource_group_member(group = aptos_framework::object::ObjectGroup)]
struct PullRewardsTransferStrategy has key {
    rewards_admin: address,
    incentives_controller: address,
    // TODO(mengxu): taking a minimal approach here but open to suggestions.
    //
    // The `rewards_vault` is essentially a resource account, marked by the
    // associated `SignerCapability`.
    //
    // Rewards will be stored in the primary `FungibleStore`s of this
    // resource account, each store is associated with a FungibleAsset.
    //
    // However, this design will need to be re-visited when we make a
    // holistic update of as many `address` to `Object` or Resource Account
    // as possible.
    rewards_vault: SignerCapability
}
```

Without any further details, documentation or specifications, there are still some unanswered questions:

1. Who and how these `SignerCapability` are generated.
2. Who can manage and have access to those `SignerCapability`.
3. Will be generated a different and unique `rewards_vault` for every reward or will be used a "common one" (following the simplification applied by Solidity).
4. How the rewards are indeed transferred to the `rewards_vault` to be later on pulled when the user claims them.

- **Problem 2: configuration and deployment of the `(asset, reward)` distributions:** From the Research performed on the Ethereum Mainnet configuration, we know that:
  - All the rewards use the same `PullTransferStrategy` contract instance.
  - All the rewards distributions will pull from the same `REWARDS_VAULT`.
  - The `REWARDS_VAULT` is the `ACI multisig`.
  - The `reward` for an `asset` could be an `AToken` and not just a "normal" `ERC20` token.

All this information relative to the Aptos deployment of the Reward System is still unknown to Spearbit and has not been documented in the codebase. For example:

1. Which type of `reward` will be used as rewards?
2. Can the `AToken` be used as a `reward` on the Aptos deployment?
3. As mentioned in the previous issue, how will be configured the `rewards_vault` and from where the rewards will be funded into the `rewards_vault`?

> **Important Note:** the current implementation of `transfer_strategy` is **incompatible** with the scenario where the `AToken` is used as a `reward`. The current implementation of the `AToken` (in the `a_token_factory`) freezes the wallets that receive the tokens (during mint and transfer operations). Both `pull_rewards_transfer_strategy_perform_transfer` and `pull_rewards_transfer_strategy_emergency_withdrawal` would revert when the token to be "pulled" is an `AToken`.

- **Problem 3: concerns relative to the "infinite" pull of funds from the `rewards_vault`:** Unlike in Solidity, where the EOA/Contract can give a finite allowance to a `spender` for a `token`, on Aptos this concept of allowance does not exist.

  This means that when you have access to the signer, or you can generate one from the `SignerCapability`, you could potentially pull all the balance from the primary storage associated with such `signer`. Aave Protocol should be aware of this possibility given that the reward system can, indeed, pull an arbitrary amount of any token owned by the primary store associated with the `SignerCapability` of the `rewards_vault`.

  The Recommendation we can provide at this stage is to ensure that the `rewards_vault` contains only the rewards needed to be later pulled by the users during the claim process.

Recommendation: Aave Protocol should:

1. Provide all the information needed to answer the question and open points described in the "Problem 1" section relative to the `rewards_vault`.
2. Provide the information needed to understand the configuration and flows of the Reward System described in the "Problem 2" section.
3. Document the concern described in the "Problem 3" section and ensure that the `rewards_vault` holds only the rewards needed.

Aave Labs: Thank you for the list of concerns and questions! We acknowledge that they are valid concerns and we would like to provide the answers to some of the explicitly raised questions here:

- **Problem 1: lack of proper explanation and documentation relative to the `rewards_vault`.**
  - Who and how these `SignerCapability` are generated.

    These `SignerCapability` is passed in as argument in the `create_pull_rewards_transfer_strategy` function which is a public function and can be accessed by a Move script. We intentionally did not specify which account the `SignerCapability` might be pointing to but in most cases, it should be a resource account that is controlled by whoever administrate the rewards. We can, alternatively, prepare an `public entry` function to do this end-to-end.
  - Who can manage and have access to those `SignerCapability`.

    Only the owner of the resource account and the `PullRewardsTransferStrategy` module.
  - Will be generated a different and unique rewards_vault for every reward or will be used a "common one" (following the simplification applied by Solidity).

    It is again, intentionally left unanswered. In this design, both one per each reward and a "common one" for all rewards are supported.
  - How the rewards are indeed transferred to the rewards_vault to be later on pulled when the user claims them.

    Normal coin/FA transfer through the Aptos Framework will do.
- **Problem 2: configuration and deployment of the (asset, reward) distributions**.
  - Which type of reward will be used as rewards?
  - Can the AToken be used as a reward on the Aptos deployment?

    With the current changes, and as updated in findings 85, both regular DFAs and ATokens can be configured as rewards. However, initially on Aptos, we will use APT.
  - As mentioned in the previous issue, how will be configured the rewards_vault and from where the rewards will be funded into the rewards_vault?

    Normal coin/FA transfer through the Aptos Framework will do.
- **Problem 3: concerns relative to the "infinite" pull of funds from the rewards_vault**.

> The Recommendation we can provide at this stage is to ensure that the rewards_vault contains only the rewards needed to be later pulled by the users during the claim process.

Acknowledged and thanks for the recommendation.

Spearbit: Thanks for the answer, we want to clarify some points about the current state of the code:

1) `AToken` can be "pulled" as rewards, but you cannot configure a reward distribution of type `(asset, AToken)` because the `oracle` module **does not** currently support `AToken` pricing. See this comment on Finding 85.
2) The pull strategy can be created **only** by users that have the `EMISSION ADMIN` role, but the distribution can only be created (and managed) by the users that have been configured as `emission_admins` for the specific `reward` in the struct `EmissionManagerData.emission_admins` of the `emission_manager` module. Those conditions are not directly connected, and you will need to configure them separately.

Aave Labs:

- Acknowledged 1, as we discussed in the linked issue.
- Acknowledged 2 as well, yes, they need to configured separately.

5.2.4 Updating/Fetching the reward's emission admin should not revert when there's no `rewards_controller` configured

Severity: Low Risk

Context: emission_manager.move#L212-L215, emission_manager.move#L249-L252

Description: The `rewards_controller` attribute stored in the `EmissionManagerData` struct of the `emission_manager` holds the value of the "current" `rewards_controller` configured. Such value could be not configured yet or simply set to `option::none()` to state the that there's no not a current reward controller in action.

The emission admin of a reward is stored as an item of `emission_admins: SmartTable<address, address>` in the `EmissionManagerData` struct is not bound to the value of `rewards_controller`. The `@aave_pool` admin user should **always** be able to configure/update an emission admin for a `reward`, even if the `rewards_controller` has not been configured yet. The same should also be true for the getter function relative to the reward's emission admin.

Recommendation: Both the `set_emission_admin` and `get_emission_admin` should not revert if the `emission_manager_data.rewards_controller` is not configured. Reward's emission admin are not bound to the value of `rewards_controller`.

Aave Labs: Fixed in PR 335.

Spearbit: Fix verified.

5.2.5 `rewards_controller` module events are not tracking which `rewards_controller_address` has emitted them

Severity: Low Risk

Context: rewards_controller.move#L130-L170

Description: The `rewards_controller` module can be seen as a "factory" of rewards controllers. Almost all the functions take an arbitrary `rewards_controller_address` address as an input parameter to distinguish which reward controller is being used for the internal function logic.

This `rewards_controller_address` address should also be included in the events that are triggered by the execution of those functions:

- `ClaimerSet`.
- `Accrued`.

- `AssetConfigUpdated`.
- `RewardsClaimed`.
- `PullRewardsTransferStrategyInstalled`.

Recommendation: Aave Protocol should include the `rewards_controller_address` input parameters in all the above events.

Aave Labs: Fixed in PR 337.

Spearbit: Fix verified.

5.2.6 Aave Core and Aave Reward use the same oracle's module

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: With the current implementation of the Aptos codebase, the `oracle` module maps both the prices of the assets used as Aave Protocol Core reserves and the one used as rewards of the Aave Periphery reward system.

This choice creates two different problems:

1) It requires using the same price source for the same token, not allowing, for example, to use the Aave CAPO for the reserve's asset and a "non-CAPO" one for the reward "flavor" of the same token.
2) It will "mix" in the oracle assets that are part of the Aave Protocol reserve with assets that won't be.

Recommendation: Aave Protocol should be aware of these requirements and limitations and consider implementing, if needed, two different oracle modules: one for the Aave Core reserve assets and one for the rewards token of the Aave Reward system.

Aave Labs: Acknowledged.

Spearbit: Acknowledged.

5.2.7 Wrong bounds check in `rewards_controller::update_reward_data`

Severity: Low Risk

Context: rewards_controller.move#L1299

Description: The `rewards_controller::update_reward_data` function wants to assert that the `new_index` fits into 104 bits (to align with the Solidity implementation).

```
assert!(
    new_index <= math_utils::pow(2, 104),
    error_config::get_ereward_index_overflow()
);
```

However, it performs a wrong bounds check, the `MAX_U104` is `2 ** 104 - 1`, not `2 ** 104`.

Recommendation:

```
  assert!(
-     new_index <= math_utils::pow(2, 104),
+     new_index < math_utils::pow(2, 104),
      error_config::get_ereward_index_overflow()
  );
```

Aave Labs: Fixed in PR 344.

Spearbit: Fix verified.

5.2.8 `EmergencyWithdrawal` spoofing

Severity: Low Risk

Context: transfer_strategy.move#L146-L151

Description: The `create_pull_rewards_transfer_strategy` function can be called by anyone to receive the `PullRewardsTransferStrategy` resource on the object given by the `constructor_ref`. This object can then be used in the other public function `pull_rewards_transfer_strategy_emergency_withdrawal` to spoof emergency withdrawals of the module.

While one cannot impersonate the legitimate `PullRewardsTransferStrategy` with the real `rewards_admin` and `rewards_vault`, one can spoof other objects created by oneself and they will all emit the `EmergencyWithdrawal` of the module.

Recommendation: Consider restricting access to the `create_pull_rewards_transfer_strategy` (preferred) or the `pull_rewards_transfer_strategy_emergency_withdrawal` function.

Aave Labs: Fixed in PR 406.

Spearbit: Fix verified.

5.2.9 `rewards_controller::handle_action` should never revert

Severity: Low Risk

Context: rewards_controller.move#L343-L346

Description: The `handle_action` function of the `rewards_controller` modules is called by the `AToken` or `VariableDebtToken` logic when the user mint/burn/transfer those tokens. Reverting during the execution of the `handle_action` function means that the above functions (in the `AToken` or `VariableDebtToken` context), that are crucial in the Aave Core logic, would break. If the `incentives_controller` attribute has not been configured (empty) or is misconfigured, the `handle_action` should just return early without tracking the user rewards and never revert.

Recommendation: If `rewards_controller_data_exists(rewards_controller_address) == false` the `handle_action` function should early return instead of reverting.

Aave Labs: Fixed in PR 346.

Spearbit: Fix verified.

5.2.10 Users could lose not-yet accrued rewards when the distribution end is updated

Severity: Low Risk

Context: rewards_controller.move#L1098

Description: The current implementation of `set_emission_per_second` is not executing `update_reward_data(...)` to update the `(asset, reward)` distribution before updating the Global Storage value for `reward_data.distribution_end`. Assuming that the distribution has not ended yet, If the new `new_distribution_end` is `<= reward_data.last_update_timestamp` all the users will lose amount of rewards that could have accrued in the delta seconds `timestamp::now_seconds() - reward_data.last_update_timestamp`.

This happens because `calculate_asset_index_internal` will early return the "old" distribution index when `last_update_timestamp >= distribution_end`.

Recommendation: Aave Protocol should trigger the calculation of the distribution index via `update_reward_data(...)` before that `reward_data.distribution_end` is updated with the new value inside `set_distribution_end`.

Aave Labs: Fixed in PR 347.

Spearbit: Fix verified.

5.2.11 `emission_manager` should expose a getter function to fetch the current `rewards_controller`

Severity: Low Risk

Context: emission_manager.move#L1

Description: The current implementation of the `emission_manager` modules does not expose a function which is the current `rewards_controller` in use. This piece of information will be important for both integrators and users when they need to interact via the dApp to claim their rewards.

Recommendation: Aave Protocol should implement a getter function that returns the current `rewards_controller` configured within the `emission_manager` module.

Aave Labs: Fixed in PR 342.

Spearbit: Fix verified.

5.3 Informational

5.3.1 `rewards_controller::initialize` should be declared `public(friend)` and executable only by the `emission_manager`

Severity: Informational

Context: rewards_controller.move#L69

Description: All the "management" functions like `set_emission_per_second`, `configure_assets` and so on of the `rewards_controller` are declared as `public(friend)` and are called by the `emission_manager` module under an auth flow check. The same concept should also be applied to the `rewards_controller::initialize` function that should be "gated" by the `emission_manager` module.

Recommendation: Aave Protocol should:

1) Declare the `rewards_controller::initialize` function as `public(friend)`.
2) Remove the `sender` check.
3) Implement a function in the `emission_manager` that will execute `rewards_controller::initialize` and declare it as `public entry`. The function should be executable only by the `@aave_pool` user.

Aave Labs: Fixed in PR 336.

Spearbit: Fix verified.

5.3.2 The oracle addresses should not be returned to the UI

Severity: Informational

Context: rewards_controller.move#L222-L225, ui_pool_data_provider_v3.move#L140

Description: Unlike in Solidity where this information makes sense to be later on be used by the UI, this information is valueless in the Aptos concept where the Aave Oracle is a pre-defined module. This makes even less sense in the case of the `reward_oracle_address` information stored in the `RewardInfo` struct returned by the `get_reserves_incentives_data` of the `ui_incentive_data_provider_v3` given that in Solidity, each `reward` returns a specific Oracle instance while here the `oracle` module address will be returned.

Recommendation: Aave Protocol should consider removing the following information unless there's a valid and specific need:

- `reward_oracle_address` from the `RewardInfo` struct of the `ui_incentive_data_provider_v3` module.
- `reward_oracle_address` from the `UserRewardInfo` struct of the `ui_incentive_data_provider_v3` module.
- `price_oracle` from the `AggregatedReserveData` struct of the `ui_pool_data_provider_v3` module.
- `get_reward_oracle` function from the `rewards_controller` module.

Aave Labs: Fixed in PR 334.

Spearbit: Fix verified.

5.3.3 The reward system should be able to use only `AToken` or `VariableDebtToken` as assets

Severity: Informational

Context: rewards_controller.move#L252

Description: The Aave Reward is implemented to assume that the `asset` configured by the `configure_assets` function from the `rewards_controller` is a valid `AToken` or `VariableDebtToken` deployed on the Aave Core system. This assumption is never enforced with an explicit sanity check.

Recommendation: The `rewards_controller::configure_assets` function should revert if the token has not been already deployed as a valid `AToken` or `VariableDebtToken`. This information can be retrieved by querying the `token_base` module.

Aave Labs: Fixed in PR 416.

Spearbit: Fix verified.

5.3.4 `collector.withdraw` does not revert if no store configured

Severity: Informational

Context: collector.move#L141

Description: The `collector.withdraw`'s comment reads:

> // check if we have a secondary fungible store for the asset, if now, throw an error.

However, it does not revert if there is no fungible store for the asset.

Recommendation: Assert that the `fungible_assets` map contains the `asset`. Typo: `if now` → `if not`.

Aave Labs: Fixed in PR 341.

Spearbit: Fix verified.

5.3.5 `is_funds_admin` and `check_is_funds_admin` naming should be swapped

Severity: Informational

Context: collector.move#L31-L41

Description: The `collector`'s `is_funds_admin` performs the check and does not return a bool, the `check_is_funds_admin` does not perform the check and returns a bool.

Recommendation: Swap the names of the functions to better align with their implementation.

Aave Labs: Fixed in PR 339.

Spearbit: Fix verified.

5.3.6 `ui_pool_data_provider_v3` is not implementing the `get_reserves_list` function

Severity: Informational

Context: ui_pool_data_provider_v3.move#L1

Description: With the requirement for the Aptos port to be 1:1 with the Solidity implementation, the `ui_pool_data_provider_v3` module should implement and expose all the functions implemented by the `UiPoolDataProviderV3` contract. With such premise, the `ui_pool_data_provider_v3` should implement and expose the `get_reserves_list` function implemented as shown in `UiPoolDataProviderV3.sol#L39-L44`.

Recommendation: Aave Protocol should implement and expose the `get_reserves_list` function in the `ui_pool_data_provider_v3` module.

Aave Labs: Fixed in PR 349.

Spearbit: Fix verified.

5.3.7 `ui_pool_data_provider_v3` should use the "full" function call version instead of the shorthand version to avoid confusion

Severity: Informational

Context: ui_pool_data_provider_v3.move#L27-L43

Description: The `ui_pool_data_provider_v3` is the only module that is importing the function's name from external modules:

```
use aave_pool::pool::{
    get_reserves_list,
    get_reserve_data,
    get_reserve_a_token_address,
    get_reserve_accrued_to_treasury,
    get_reserve_current_liquidity_rate,
    get_reserve_current_variable_borrow_rate,
    get_reserve_isolation_mode_total_debt,
    get_reserve_last_update_timestamp,
    get_reserve_liquidity_index,
    get_reserve_unbacked,
    get_reserve_variable_borrow_index,
    get_reserve_variable_debt_token_address,
    get_reserve_configuration_by_reserve_data,
    get_user_configuration,
    get_reserve_id
};
```

to then be called without prefixing them with the module's name when used. This creates two problems:

1. It breaks the code style already adopted by all the other codebase of the project.
2. It makes the code less readable. It's harder to understand at first sight if the function is a "local" defined one or if it's an external function imported from another module.

Recommendation: Aave Protocol should use the "full version" of those function invoking them by prefixing the function name with the module's name.

Aave Labs: Fixed in PR 350.

Spearbit: Fix verified.

5.3.8 `get_reserves_data::ui_pool_data_provider_v3` should use the specific token factory functions and not `token_base` ones

Severity: Informational

Context: ui_pool_data_provider_v3.move#L159-L160

Description: The `total_scaled_variable_debt` variable in `get_reserves_data` is calculated as `token_base::scaled_total_supply(variable_debt_token_address)`. While there's no security issue involved, the code would result easier to read and more correct if it was invoking the same function but from the `variable_debt_token_factory` module, given that the `variable_debt_token_address` used is indeed a `VariableDebtToken` token.

Recommendation: Aave Protocol should use the more specific `variable_debt_token_address` module instead of the underlying `token_base` one.

Aave Labs: Fixed in PR 351.

Spearbit: Fix verified.

5.3.9 `base_currency` refactoring and simplification

Severity: Informational

Context: oracle_base.move#L87, oracle_base.move#L92-L110, ui_pool_data_provider_v3.move#L45-L46, ui_pool_data_provider_v3.move#L275-L284

Description: From the information we were able to gather, we can make this assumption: the `BASE_CURRENCY` information on Aave Oracle is used to represent the `USD` currency with `8` decimals.

On Solidity the `BASE_CURRENCY` on Aave Oracle is represented as an `immutable` state variable that can be configured during the Aave Oracle deployment but has been consolidated, as mentioned above, as the 1 unit of USD with 8 decimals.

Given this assumption, we can simplify and refactor the Aptos codebase everywhere the `base_currency` is used.

- The `base_currency: Option<BaseCurrency>` attribute from the `PriceOracleData` struct can inside the `oracle_base` module can be removed with all the functions and usage inside the module itself.
- `ui_pool_data_provider_v3` can be refactored with these changes:

```
- let opt_base_currency = oracle_base::get_oracle_base_currency();
- let (market_reference_currency_unit, market_reference_currency_price_in_usd) =
-     if (option::is_some(&opt_base_currency)) {
-         let unit =
-             oracle_base::get_base_currency_unit(
-                 &*option::borrow(&opt_base_currency)
-             );
-         ((unit as u256), (unit as u256))
-     } else {
-         (APT_CURRENCY_UNIT, oracle::get_asset_price(apt_mapped_fa_asset))
-     };
+ let market_reference_currency_unit = USD_CURRENCY_UNIT;
+ let market_reference_currency_price_in_usd = USD_CURRENCY_UNIT;
```

With `USD_CURRENCY_UNIT` defined as `const USD_CURRENCY_UNIT: u256 = 1_000_000_000_000_000_000;` given that Chainlink by default uses `18` decimals for their prices. Both the `EMPTY_ADDRESS` and `APT_CURRENCY_UNIT` constant variables can be **removed**.

> **Note:** it's still unclear if other modules inside the Aave Protocol (that we are not aware of), integrators modules or dApps need the "direct" access to the `base_currency` information from the Aave Oracle. If that's the case, Aave Protocol will need to provide direct access to it or simply ask them to hardcode it as a constant given the above assumptions.

Recommendation: If the assumption made are valid, Aave Protocol should refactor and simplify all the code and logic that is based on the `base_currency` concept as explained above.

Aave Labs: Fixed in PR 338.

Spearbit: Fix verified.

5.3.10 `rewards_controller` should only use `rewards_controller_data_exists` for consistency

Severity: Informational

Context: rewards_controller.move#L766-L768, rewards_controller.move#L927-L929, rewards_controller.move#L956-L958, rewards_controller.move#L1011-L1013, rewards_controller.move#L1436-L1438

Description: The `rewards_controller_data_exists` function in the `rewards_controller` module checks the existence of the `RewardsControllerData` global storage struct for the `rewards_controller_address`.

```
fun rewards_controller_data_exists(
    rewards_controller_address: address
): bool {
    exists<RewardsControllerData>(rewards_controller_address)
}
```

but part of the codebase does not leverage this function and manually check it directly in the `if` condition.

Recommendation: To be consistent with the existing code and best practice, Aave Protocol should replace any part of the codebase that directly checks the existence of `RewardsControllerData` for `rewards_controller_address` with a call to the `rewards_controller_data_exists` function.

Aave Labs: Fixed in PR 348.

Spearbit: Fix verified.