SHERLOCK

Security Review For Canopy

Collaborative Audit Prepared For:	**Canopy**
Lead Security Expert(s):	**JakubHeba**
	sh1v
Date Audited:	**September 3 - September 17, 2025**

Introduction

Canopy is the gateway to Movement DeFi built to optimize liquidity management through an all-in-one yield aggregation platform.

Canopy can be understood by its four key layers:

- Application Layer: User-friendly interface for effortless deployment of yield generating liquidity.
- Reward Distribution Engine: Enables any project to incentivize value creating behavior with $MOVE or their project token.
- Strategy Layer: Smart contracts that define optimal liquidity flows across the Movement network.
- Protocol Layer: Integrated AMMs, lending platforms, perpetual DEXs (PerpDEXs), and staking platforms.

Together, these layers form a platform where liquidity providers thrive.

Scope

Repository: Canopyxyz/ichi-vaults-thala

Audited Commit: b869d94732a9f52676f825778b94146c5711a9c0

Final Commit: 995aba18d8e6adb61b74e942df872efa703353ed

Files:

- packages/ichi-vaults/sources/libs/math_helpers.move
- packages/ichi-vaults/sources/libs/pool_helpers.move
- packages/ichi-vaults/sources/libs/timelocked.move
- packages/ichi-vaults/sources/router.move
- packages/ichi-vaults/sources/vault.move

Final Commit Hash

995aba18d8e6adb61b74e942df872efa703353ed

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	2	10

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: An u64 overflow risk in distribute_single_asset_fee[FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/44

Summary

The distribute_single_asset_fee multiplies total * fee_bps_2 * split_bps using u64 operands, which can overflow before division even when the final fee amount would fit into u64.

Vulnerability Detail

In Move, arithmetic happens in the type of the operands. The function multiplies three u64 values: the token total, which may represent large balances with 6-8+ decimals, fee_bps_2, and a per-recipient split_bps.

It is realistic that the result can exceed u64::MAX and abort the transaction before division by the basis-point scalars. This is problematic, as this function is used by rebalance, fee claiming and emergency exit flows.

Impact

At sufficient TVL or with moderately high BPS inputs, rebalances that trigger fee distribution can revert and effectively DoS strategy operations. The vault may become unable to rotate positions or claim fees until parameters or balances are reduced.

Code Snippet

```
if (total > 0) {
            let amount = (total * fee_bps_2 * split_bps) / (BPS_PRECISION *
             ↪  BPS_PRECISION * BPS_PRECISION);
            if (amount > 0) {
                primary_fungible_store::deposit(recipient,
                 ↪  fungible_asset::extract(asset, amount));
            };
        };
```

Tool Used

Manual Review

Recommendation

Use `u128` types to do staged mul and div so no intermediate exceeds `u128`.

Issue M-2: The `add_liquidity_to_position` always mints a new position NFT [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/48

Summary

When `find_existing_position` detects a matching (lower, upper) range, `add_liquidity_to_position` still mints a new position via `pool::new_position` and pushes it to `positions`. The path never reuses the existing position token for that range.

Vulnerability Detail

The `add_liquidity_to_position` function's structure checks for an existing mapping entry but proceeds to mint a new position regardless, only conditionally updating the map if absent.

This behavior leads to multiple NFTs for the same price range owned by the vault, fragmenting liquidity and growing positions unboundedly. It also risks drift between `position_map` and the positions vector.

Impact

Liquidity fragmenting, storage growth and higher gas costs due to many small position objects.

Code Snippet

```
let token_obj_option = find_existing_position(vault_ref, lower, upper);

        let asset_0_amount = fungible_asset::amount(asset_0);
        let asset_to_add_0 = fungible_asset::extract(asset_0, asset_0_amount);
        let asset_1_amount = fungible_asset::amount(asset_1);
        let asset_to_add_1 = fungible_asset::extract(asset_1, asset_1_amount);
        // add_liquidity will return any remaining_{0,1} that was not needed for
        ↪  the specified liquidity to mint
        // so there's no issue passing the entire asset_{0,1}
        let (token_obj, remaining_0, remaining_1) =
            pool::new_position(
                signer::address_of(&object::generate_signer_for_extending(&vault_re
                ↪  f.extend_ref)),
                vault_ref.underlying_pool,
                liquidity,
                asset_to_add_0,
                asset_to_add_1,
                lower,
                upper
            );
```

Tool Used

Manual Review

Recommendation

If `find_existing_position` locates a position for (lower, upper), call the pool's `add_liquidity` function for that token instead of minting a new one.

Issue L-1: The `set_rebalance_paused` authorization is inconsistent with the comment [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/43

Summary

The `set_rebalance_paused` uses `assert_rebalancer_or_owner`, while the inline comment states "Only the vault owner can change this setting." The behavior and the documentation contradict each other.

Vulnerability Detail

Operational tooling and governance expectations may rely on the comment ("owner-only"), but the function currently authorizes both the designated rebalancer and the owner to set the `is_rebalance_paused`. This is inconsistent and might be misleading for readers and logic, if comment is the expected approach.

Impact

A rebalancer can pause or unpause rebalancing unexpectedly, leading to denial-of-service scenario if paused at a critical time, or forced execution if unpaused against owner wishes. This can interfere with safety controls that depend on owner-gated actions.

Code Snippet

```
/// Pauses or unpauses rebalancing for the vault.
    /// Only the vault owner can change this setting.
    public entry fun set_rebalance_paused(account: &signer, vault: Object<Vault>,
     ↪  paused: bool) acquires Vault {
        let vault_ref = borrow_global_mut<Vault>(object::object_address(&vault));
        assert_rebalancer_or_owner(vault_ref, account);

        // Only proceed if the state is actually changing
        if (vault_ref.vault_config.is_rebalance_paused != paused) {
            let previous_state = vault_ref.vault_config.is_rebalance_paused;
            vault_ref.vault_config.is_rebalance_paused = paused;

            // Emit event for the state change
            event::emit(RebalancePausedEvent { vault_address:
             ↪  object::object_address(&vault), previous_state, new_state: paused
             ↪  });
        }
    }
```

Tool Used

Manual Review

Recommendation

Decide on the intended policy and align code and docs:

- If the function has to be owner only - replace assert_rebalancer_or_owner with an owner check and add tests.
- If rebalancer should have this power - update the comment accordingly

Issue L-2: “Locked” shares can be recovered via `recover_assets`[FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/45

Summary

At genesis the vault mints `INITIAL_LOCKED_SHARES` to its own primary fungible store. The `recover_assets` owner function allows withdrawing any non-pool asset from that store, including the vault’s own share token, making the “lock” unenforceable.

Vulnerability Detail

The expected approach is to permanently lock tranche of shares to stabilize early share price. However, because `recover_assets` only excludes the pool assets, the owner can recover the vault share token metadata and withdraw those “locked” shares into an external account. Those shares can then be burned in withdraw, extracting underlying assets.

Impact

The owner or a compromised owner key can redeem a portion of TVL that users expect to be non-redeemable. Even if disclosed, it is materially different from a true lock.

Code Snippet

```
public fun recover_assets(
        account: &signer, vault: Object<Vault>, fungible_asset_metadata:
        ↪  Object<Metadata>
    ): FungibleAsset acquires Vault {
        // NOTE: we shouldn't need to coin::migrate_to_fungible_store
        // since the vault never exposed it's signer to ever have a legacy coin
        ↪  balance
        let vault_ref = borrow_global_mut<Vault>(object::object_address(&vault));

        // Assert caller is vault owner
        assert_signer_is_vault_owner(vault_ref.owner, account);

        // Get metadata for pool assets and vault shares
        let (metadata_0, metadata_1) =
        ↪  pool::pool_metadata(vault_ref.underlying_pool);

        // Ensure we're not trying to recover pool assets(No issue recovering vault
        ↪  shares)
        assert!(
            fungible_asset_metadata != metadata_0 && fungible_asset_metadata !=
            ↪  metadata_1,
            EINVALID_RECOVERY_ASSET
        );
```

Tool Used

Manual Review

Recommendation

Explicitly forbid the share token metadata in `recover_assets` and related batch functions, or mint `INITIAL_LOCKED_SHARES` to an unspendable account, for example a separate store not covered by recovery paths.

Discussion

mshakeg

this was a deliberate decision by us as we don't see any issue with allowing those ”locked” shares to be recovered

jakub-heba

There is an attack vector called First Depositor Inflation Attack (https://mixbytes.io/blog/overview-of-the-inflation-attack). Based on that, standard security assumption is that these shares cannot be withdrawn by any party, even privileged one.

While I understand your concerns, we would like to have this issue reflected in the report with Acknowledged status.

mshakeg

@jakub-heba we've decided to allow vault share recovery up to `INITIAL_LOCKED_SHARES`

https://github.com/Canopyxyz/ichi-vaults-thala/pull/9

jakub-heba

Looks good!

Issue L-3: Rebalance lacks built-in slippage protection [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/46

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The rebalance flow does not enforce on-chain slippage protections. If `sqrt_price_limit` is 0, the swap helpers use extreme boundary price limits and `min_amount_out` is not enforced at the vault layer. Without both a reasonable `sqrt_price_limit` and non-zero `min_amount_out`, the vault is exposed to slippage and price-manipulation risk.

Vulnerability Detail

While router paths may include pre and post tick checks, the core rebalance entry points accept parameters that allow boundary-price swaps when `sqrt_price_limit == 0`. Because `min_amount_out` is not verified inside the vault, an authorized rebalancer can execute swaps with effectively unbounded slippage.

Impact

A single rebalance can push trades to pool extremes, converting assets at highly unfavorable prices and inflicting permanent loss on the vault. This risk increases under volatile markets or if the rebalancer key is compromised.

Code Snippet

```
public fun rebalance(
       account: &signer,
       vault: Object<Vault>,
       new_positions: vector<RebalancePosition>,
       amount_in: u64,
       zero_for_one: bool,
       sqrt_price_limit: u128
   ) acquires Vault, GlobalVaultConfig {
```

Tool Used

Manual Review

Recommendation

Require non-zero `min_amount_out` at the vault layer and verify it.

Discussion

mshakeg

sounds similar to #56

the `vault::rebalance` function has a hysteresis check to ensure that the current spot price is close to recent TWAPs, though beyond this the caller should generally specify a valid swap price limit.

jakub-heba

You're right that the hysteresis check restrict extreme market moves, but it's not slippage protection itself - rebalance can still execute at an extreme price if a rebalancer is misconfigured or compromised. We'll suggest to keep this finding, even if the risk itself will be acknowledged.

Issue L-4: The `EmergencyExitEvent` event `removed_positions_count` always reports zero [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/47

Summary

The `EmergencyExitEvent.removed_positions_count` is populated after clearing positions, so it will always emit 0 even if positions were removed.

Vulnerability Detail

The `EmergencyExitEvent` event constructs `removed_positions_count` from the length of `vault_ref.positions` after all liquidity has been removed and the vector cleared. This produces misleading data for monitoring and post-incident analysis.

Impact

Operators and auditors cannot rely on the event to understand how many positions were unwound during an emergency. This hardens incident response and automated alerting.

Code Snippet

```
event::emit(
            EmergencyExitEvent {
                vault_address: object::object_address(&vault),
                executor: signer::address_of(account),
                removed_positions_count: vector::length(&vault_ref.positions),
                collected_fees_0: amount_collected_fees_0,
                collected_fees_1: amount_collected_fees_1,
                removed_0: amount_removed_0,
                removed_1: amount_removed_1
            }
        );
```

Tool Used

Manual Review

Recommendation

Read and store ”length before” from positions prior to the removal loop.

Issue L-5: Inconsistent validation between `manage_timelocked_deposit_twap_period` and `manage_timelocked_aux_deposit_twap_period` [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/49

Summary

The `manage_timelocked_deposit_twap_period` validates the TWAP period, like min and max bounds, but the `manage_timelocked_aux_deposit_twap_period` omits this.

Vulnerability Detail

The vault exposes two management functions to set TWAP windows used during deposit valuation - a primary control `manage_timelocked_deposit_twap_period` and an auxiliary one - `manage_timelocked_aux_deposit_twap_period`. The primary setter invokes a validation routine calling `assert_valid_time_period` that enforces reasonable bounds on the sampling window.

The auxiliary setter lacks this validation and accepts arbitrary values, including 0. Because deposit valuation code paths can consult the auxiliary window under specific configurations, an operator can set the auxiliary period down to 0, effectively collapsing TWAP to spot price and weakening the vault's intended anti-volatility and anti-manipulation defenses. This asymmetry also increases configuration complexity as the system may appear guarded while the auxiliary path silently disables those protections.

Impact

Deposit valuation may rely on insufficiently robust observations when the auxiliary period is reduced below safe thresholds, increasing susceptibility to short-term price manipulation.

Code Snippet

```
public entry fun manage_timelocked_aux_deposit_twap_period(
        account: &signer,
        vault: Object<Vault>,
        new_aux_deposit_twap_period: Option<u64>,
        new_delay: Option<u64>
    ) acquires Vault {
        let vault_ref = borrow_global_mut<Vault>(object::object_address(&vault));
        assert_signer_is_vault_owner(vault_ref.owner, account);
        timelocked::manage_timelocked_update(
            &mut vault_ref.vault_config.timelocked_aux_deposit_twap_period,
             ↪  new_aux_deposit_twap_period, new_delay
        );
    }
```

Tool Used

Manual Review

Recommendation

Invoke `assert_valid_time_period` in `manage_timelocked_aux_deposit_twap_period` as well.

Issue L-6: The `cancel_update` does not emit an event [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/50

Summary

The `cancel_update` action currently makes a state change without emitting an event, unlike related management actions that do emit events.

Vulnerability Detail

Timelocked parameter management actions generally emit events that document what was changed and when the change becomes effective. The `cancel_update` action, which aborts a pending change before it matures, currently performs a stateful cancel but emits no event. This leaves a blind spot for off-chain systems that track configuration drift or alert on cancelled changes.

Impact

incomplete audit trail for cancelled parameter updates, which complicates compliance and debugging.

Code Snippet

```
public fun cancel_update<T: copy + drop + store>(locked: &mut TimeLocked<T>) {
        // Only proceed if there is a pending update to cancel
        if (option::is_some(&locked.pending_update)) {
            // Clear all pending state
            option::extract(&mut locked.pending_update);
        }
    }
```

Tool Used

Manual Review

Recommendation

Emit a `CancelUpdateEvent` when `cancel_update` is called, or make `manage_timelocked_update` the sole entry point and include a field like `action: UPDATE | CANCEL | RESCHEDULE` so cancellations are observable from a single event stream.

Issue L-7: The `can_rebalance` return code documentation mismatch [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/51

Summary

The comment for `can_rebalance` states that `REBALANCE_SUCCESS` code is 0, but the constant used by the function is 100. The function returns these status codes to callers.

Vulnerability Detail

Off-chain bots and dashboards may implement logic based on the documented value (0) and misinterpret a 100 return as a failure (or vice versa). This can cause missed rebalances or repeated attempts.

Impact

Automation instability and potential strategy downtime if bots skip valid opportunities or loop on supposed failures. This also risks noisy alerts and operator mistakes.

Code Snippet

```
 // - - - REBALANCE CHECKS CODES - - -

    const REBALANCE_SUCCESS: u64 = 100;
[..]
#[view]
    /// Checks if a rebalance operation would be possible given current conditions.
    /// Returns a tuple (bool, u64) where:
    /// - bool: true if rebalance is possible, false otherwise
    /// - u64: error code indicating the reason for failure, or REBALANCE_SUCCESS
    ↪  (0) if successful
    ///   Possible error codes:
    ///   - REBALANCE_ERROR_PAUSED (101): Rebalancing is paused
    ///   - REBALANCE_ERROR_NOT_AUTHORIZED (102): Address is not authorized to
    ↪ rebalance
    ///   - REBALANCE_ERROR_TOO_MANY_POSITIONS (103): Too many positions specified
    ///   - REBALANCE_ERROR_EXCESSIVE_VOLATILITY (104): Market volatility exceeds
    ↪ threshold
    public fun can_rebalance(vault: Object<Vault>, rebalancer_addr: address,
    ↪ new_positions_count: u64): (bool, u64) acquires Vault {
        let vault_ref = borrow_global<Vault>(object::object_address(&vault));
        check_rebalance_possible(vault_ref, rebalancer_addr, new_positions_count)
    }
```

Tool Used

Manual Review

Recommendation

Set `REBALANCE_SUCCESS` code to 0 or update the comment and constants list to say that success code is 100.

Issue L-8: The `get_user_vault_balance` view function attempts to create a primary store [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/52

Summary

The `get_user_vault_balance` is annotated as a `#[view]` but calls `primary_fungible_store::ensure_primary_store_exists`. If the user has no store, the function attempts to create one, which is a state mutation not permitted in a `view` function.

Vulnerability Detail

The `get_user_vault_balance` is annotated `#[view]` and is expected to be read-only. Internally, it calls `primary_fungible_store::ensure_primary_store_exists` for the vault share metadata on the target address. That helper is stateful so if the address has never held the share token before, it attempts to create a primary store under that account.

Aptos enforces read-only semantics for `#[view]` functions, so any invocation that would create the store reverts instead of returning a balance, making the getter unreliable for first-time users and breaking indexers that want a uniform "zero if absent" read. The result is that dashboards or wallets calling the view against addresses that have not yet interacted with the share will intermittently fail, depending on whether the store exists, violating the principle that view calls should have no side effects and should succeed across the entire state space.

Impact

Partial unavailability of the getter, breaking indexers, frontends, or wallet integrations that rely on `get_user_vault_balance` to display "zero" balances for first-time users.

Code Snippet

```
#[view]
    public fun get_user_vault_balance(vault: Object<Vault>, user: address): (u64,
     ↪  u128) acquires Vault {
        let vault_ref = borrow_global<Vault>(object::object_address(&vault));

        let user_shares =
            fungible_asset::balance(primary_fungible_store::ensure_primary_store_ex⌋
             ↪  ists(user, vault_ref.vault_shares_metadata));

        if (user_shares == 0) {
            return (0, 0)
        };

        // Get the value of these shares in the deposit asset
        let share_price_e18 = get_shares_price_e18(vault);
        let value = math128::mul_div(share_price_e18, (user_shares as u128),
        ↪  E18_PRECISION);

        (user_shares, value)
    }
```

Tool Used

Manual Review

Recommendation

Replace `ensure_primary_store_exists` with a safe existence check and read-only query which return zero if absent.

Discussion

mshakeg

I've instead used `primary_fungible_store::balance` in `vault::get_user_vault_balance`

jakub-heba

Looks good!

Issue L-9: Vault rebalance stores lack transfer restrictions [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/55

Summary

The vault's rebalance stores do not have transfer restrictions explicitly disabled, creating a potential security risk if framework changes affect transfer behavior in the future.

Vulnerability Detail

The vault creates managed fungible stores for rebalancing without calling `object::disable_ungated_transfer()`. While currently secure due to resource account ownership, this lacks explicit transfer restrictions that would prevent stores from being transferred regardless of underlying asset metadata configuration or future framework changes.

Impact

While current implementation is secure via resource account ownership, missing explicit transfer restrictions could pose risks if Aptos framework behavior changes.

Code Snippet

```
fun create_managed_fungible_store(vault_signer: &signer, metadata:
 ↪  Object<Metadata>): ManagedFungibleStore {
        let store_constructor_ref =
         ↪  object::create_object(signer::address_of(vault_signer));
        let store = fungible_asset::create_store(&store_constructor_ref, metadata);
        let store_extend_ref = object::generate_extend_ref(&store_constructor_ref);
        // Better to add explicit transfer restrictions to the object to make sure
         ↪  that this store cannot be transferred.
        ManagedFungibleStore { store, store_extend_ref }
    }
```

Tool Used

Manual Review

Recommendation

Add `object::disable_ungated_transfer()` call in `create_managed_fungible_store` to explicitly disable store transfers. This defensive measure aligns with Aptos framework's approach for deterministic stores and ensures vault stores remain permanently bound regardless of future changes.

Issue L-10: TWAP Period Misconfiguration in Rebalance [FIXED]

Source: https://github.com/sherlock-audit/2025-09-canopy-alm-sept-3rd/issues/58

Vulnerability Detail

The auxiliary TWAP period for rebalance volatility validation is configured as 5 seconds instead of the intended 30 seconds as indicated by code comments.

Tool Used

Manual Review

Recommendation

Update the constant to match the intended 30-second period or update the comment.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.