plainshift

July 14th, 2025

Full Sail Governance

Comprehensive Security Assessment

contact@plainshift.io
plainshift.io

Table of Contents

Disclaimer

This security assessment represents a time-boxed security review using tooling and manual review methodologies. Our findings reflect our comprehensive evaluation of the materials provided in-scope and are specific to the commit hash referenced in this report.

The scope of this security assessment is strictly limited to the code explicitly specified in the report. External dependencies, integrated third-party services, libraries, and any other code components not explicitly listed in the scope have not been reviewed and are excluded from this assessment.

Any modifications to the reviewed codebase, including but not limited to smart contract upgrades, protocol changes, or external dependency updates will require a new security assessment, as they may introduce considerations not covered in the current review.

In no event shall Plainshift's aggregate liability for all claims, whether in contract or any other theory of liability, exceed the Services Fee paid for this assessment. The client agrees to hold Plainshift harmless against any and all claims for loss, liability, damages, judgments and/or civil charges arising out of exploitation of security vulnerabilities in the contracts reviewed.

By accepting this report, you acknowledge that deployment and implementation decisions rest solely with the client. Any reliance upon the information in this report is at your own discretion and risk. This disclaimer is governed by and construed in accordance with the laws specified in the engagement agreement between Plainshift and the client.

1. Introduction

Plainshift is a full-stack security firm built on the “shift left” security philosophy. We often work with teams early in the product development process to bring security to a greater organizational range than just smart contracts. From the web app, to fuzzing/formal verification, to a team’s operational security, full-stack security can only be achieved by first understanding there is no “scope” to fully protect the users that trust you.

We’re here to meaningfully revolutionize how teams approach security and guide them towards a holistic approach rather than the single sided approach so prevalent today.

Learn more about us at https://plainshift.io.

1.1. Executive Summary

Plainshift was tasked with reviewing the Full Sail Governance contracts (distribution and liquidity locker v2) from June 24th to July 15th, 2025. We ultimately found and confirmed 1 critical, 5 high, 8 medium, 4 low severity issues and 5 informational findings.

Following a thorough audit and detailed drafts of potential attack vectors, we set up a custom testing suite to verify PoCs for our leads.

We should note that, as per our account, the in-scope codebase is quite complex, having in mind the expected business logic, and during our audit multiple issues were found, directly influencing critical protocol flows.

1.2. Project Timeline

Date	Phase
June 24th, 2025	Kickoff
July 15th, 2025	Audit End
July 15th, 2025	Delivery

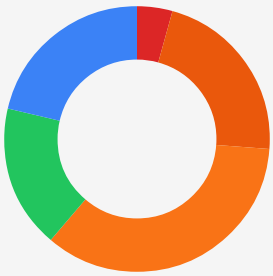
1.3. Scope

Repositories	https://github.com/LFBuild/FullSail-SC
Version	`4dc036a9a04c31fb689682821572c21916b50c7a`
Contracts	`distribution/sources/*.move`, `liquidity_locker/sources/liquidity_lock_v2.move`
Type	Move
Platform	Sui

1.4. Overview of Findings

Our comprehensive review yielded 1 critical, 5 high, 8 medium, 4 low severity issues alongside 5 informational findings.

Severity/Impact Level	Count
Critical	1
High	5
Medium	8
Low	4
Informational	5



2. Findings

2.1. [C1] Missing Emergency Council Creation Function Prevents Protocol Safety Mechanism

Target	distribution/sources/emergency_council.move
Severity	● Critical
Category	Vulnerability

2.1.1. Description

The `emergency_council` module is designed to provide critical safety controls for the protocol, allowing immediate response to security threats, exploits, or malfunctioning components. However, the module completely lacks a public function to create the `EmergencyCouncilCap` object. The only creation function available is `create_for_testing()` which is marked with `#[test_only]`, making it unavailable in production deployments. This means there is no way to actually instantiate an emergency council in the live protocol.

2.1.2. Impact

This omission completely disables the protocol's emergency response capabilities, leaving it vulnerable to various critical scenarios that require immediate intervention. The `EmergencyCouncilCap` is specifically designed to enable the following emergency actions, none of which can be performed without it:

- Kill gauges (deactivate pools) in case of exploits or vulnerabilities
- Revive previously killed gauges once issues are resolved
- Deactivate managed locks if compromised or being misused
- Execute other emergency safety measures when regular governance would be too slow

2.1.3. Recommendation

We recommend adding a public function to create the `EmergencyCouncilCap`, likely restricted to be callable only by the intended parties.

2.2. [H1] Missing `EmergencyCouncilCap` Validation in `set_managed_lock_deactivated`

Target	distribution/sources/voting_escrow.move
Severity	High
Category	Vulnerability

2.2.1. Description

The `set_managed_lock_deactivated` entry point takes an `EmergencyCouncilCap` but never calls any of its `validate_*` functions:

```
public fun set_managed_lock_deactivated<SailCoinType>(
    voting_escrow: &mut VotingEscrow<SailCoinType>,
    _emergency_council_cap: &distribution::emergency_council::EmergencyCouncilCap,
    lock_id: ID,
    deactivated: bool
) {
    assert!(voting_escrow.escrow_type(lock_id) == EscrowType::MANAGED,
    ESetManagedLockNotManagedType);
    assert!(
        !voting_escrow.deactivated.contains(lock_id) ||
        voting_escrow.deactivated.borrow(lock_id) != &deactivated,
         ESetManagedLockAlreadySet
     );
     if (voting_escrow.deactivated.contains(lock_id)) {
         voting_escrow.deactivated.remove(lock_id);
     };
     voting_escrow.deactivated.add(lock_id, deactivated);
 }
```

Because the cap is never validated, any `EmergencyCouncilCap`, even one issued for a different `VotingEscrow` instance, can be used to pause or unpause managed locks here.

2.2.2. Impact

The issue breaks the concept of pause mechanism, as a malicious cap holder can freeze or unfreeze any managed lock in any escrow.

2.2.3. Recommendation

We recommend adding both cap validation calls at the top of the function, that is `validate_emergency_council_voter_id` and `validate_emergency_council_minter_id`.

2.3. [H2] Bypass of manager lock deactivation on deposit mechanism

Target	distribution/sources/voting_escrow.move
Severity	High
Category	Vulnerability

2.3.1. Description

Even after a managed lock has been deactivated via `set_managed_lock_deactivated`, users can still deposit new tokens into it because `deposit_for` entrypoint does not check `deactivated` flag.

Since this method does not check `voting_escrow.deactivated(lock_id)`, an attacker can still call `deposit_for` on a deactivated managed lock, bypassing the intended pause.

That means that deactivating a managed lock has no effect on this public entry point, even though `deposit_managed` itself correctly checks `deactivated`.

```
public fun deposit_for<SailCoinType>(
    voting_escrow: &mut VotingEscrow<SailCoinType>,
    lock: &mut Lock,
    coin: sui::coin::Coin<SailCoinType>,
    clock: &sui::clock::Clock,
    ctx: &mut TxContext
) {
    let deposit_amount = coin.value<SailCoinType>();
    voting_escrow.balance.join<SailCoinType>(coin.into_balance());
     voting_escrow.increase_amount_for_internal(
         object::id<Lock>(lock),
         deposit_amount,
         DepositType::DEPOSIT_FOR_TYPE,
         clock,
         ctx
     );
     lock.amount = lock.amount + deposit_amount;
 }
```

2.3.2. Impact

Emergency paused managed positions can be topped up, breaking the entire deactivation flow and enabling unauthorized token depositing.

2.3.3. Recommendation

We recommend inserting a deactivation guard at the top of `deposit_for` (or in `increase_amount_for_internal`), so that any deposit into a paused managed lock will revert.

2.4. [H3] Poke Function Cannot Be Called Permissionlessly Due to Lock Object Requirement

Target	distribution/sources/voter.move
Severity	High
Category	Vulnerability

2.4.1. Description

The `poke` function in the voter module is designed to allow anyone to update a user's voting allocation based on their current voting power, preventing users from receiving rewards based on outdated (higher) voting power as their locks decay over time.

However, the function requires a reference to the user's `Lock` object as a parameter, which can only be provided by the lock owner in Sui's object model. This makes the function inaccessible to third parties, defeating its intended purpose of permissionless voting power updates, creating a scenario where inactive users maintain inflated voting power because no one else can update their stale votes.

```
/// "Pokes" the voting system to update a lock's votes based on its current voting
power.
/// This is useful when a lock's voting power changes and votes need to be recalculated.
///
/// # Arguments
/// * `voter` - The voter contract reference
/// * `voting_escrow` - The voting escrow reference
/// * `distribution_config` - The distribution configuration
/// * `lock` - The lock to update votes for
/// * `clock` - The system clock
 /// * `ctx` - The transaction context
 public fun poke<SailCoinType>(
     voter: &mut Voter,
     voting_escrow: &mut distribution::voting_escrow::VotingEscrow<SailCoinType>,
     distribution_config: &distribution::distribution_config::DistributionConfig,
     lock: &distribution::voting_escrow::Lock,   //@audit should pass ID
     clock: &sui::clock::Clock,
     ctx: &mut TxContext
 ) { }
```

2.4.2. Impact

This vulnerability allows users to maintain disproportionate voting power and receive excess rewards without any action on their part, receiving more rewards than they should based on their actual voting power.

2.4.3. Recommendation

We recommend modifying the `poke` function to accept a lock ID instead of a `Lock` object reference

```
public fun poke<SailCoinType>(
    voter: &mut Voter,
    voting_escrow: &mut distribution::voting_escrow::VotingEscrow<SailCoinType>,
    distribution_config: &distribution::distribution_config::DistributionConfig,
-   lock: &distribution::voting_escrow::Lock,
+   lock: ID,
     clock: &sui::clock::Clock,
     ctx: &mut TxContext
) { }
```

2.5. [H4] Unit mismatch in delegation cooldown check

Target	distribution/sources/voting_escrow.move
Severity	High
Category	Vulnerability

2.5.1. Description

In `distribution` module `delegate_internal`, the code enforces a cooldown after an ownership change via:

```
assert!(
    clock.timestamp_ms() - *voting_escrow.ownership_change_at.borrow(
        lock_id
    ) >= distribution::common::get_time_to_finality(),
    EDelegateOwnershipChangeTooRecent
);
```

However, `clock.timestamp_ms` is in milliseconds, but `get_time_to_finality` returns a duration in seconds.

```
/// Returns the time required for transaction finality
///
/// # Returns
/// The time in seconds required for transaction finality (500)
public fun get_time_to_finality(): u64 {
    500
}
```

Having in mind, that `get_time_to_finality` is hardcoded to 500 seconds, user will be able to redelegate almost immediately, as this value will be reduced 1000 times (0.5s).

2.5.2. Impact

Malicious actors could spam delegate calls, bloating onchain activity and potentially performing Denial of Service attack on light clients or upstream indexers expecting lower call volumes.

2.5.3. Recommendation

We recommend adjusting the assert to use miliseconds or seconds across all calculations.

2.6. [H5] Denial of Service via unreduced tranche volume

Target	liquidity_locker/sources/liquidity_lock_v2.move
Severity	High
Category	Vulnerability

2.6.1. Description

In `liquidity_lock_v2.move`, the `fill_tranches` function from `pool_tranche.move` module is invoked whenever a new lock deposits volume into a tranche through the `lock_position`, and it correctly increments `tranche.current_volum` and sets `tranche.filled` when capacity is reached.

However, there is no corresponding logic to decrement `current_volume` or clear the `filled` flag when positions are later unlocked or liquidity is removed. As a result, once a tranche ever becomes "filled", it remains permanently closed, even if users subsequently withdraw their funds.

Over time, every tranche flips into this irreversibly "filled" state, and no further locks can be accepted despite actual capacity freeing up.

2.6.2. Impact

After enough lock and unlock cycles, all tranches will be marked full, blocking any new positions and effectively halting the locking protocol.

2.6.3. Recommendation

We recommend introducing a mirror "unfill" operation in the unlock and withdrawal paths to subtract freed volume and clear the filled flag when below capacity.

2.7. [M1] Uncapped loop Denial of Service in reward claiming flow

Target	distribution/sources/reward.move
Severity	Medium
Category	Vulnerability

2.7.1. Description

The `earned` function in `reward.mov`e, called through, for example, `claim_voting_fee_reward`, iterates through every epoch since the last claim without any cap:

```
let mut next_epoch_time = latest_epoch_time;
let epochs_until_now = (distribution::common::epoch_start(
    distribution::common::current_timestamp(clock)
) - latest_epoch_time) / distribution::common::week();
if (epochs_until_now > 0) {
    let mut i = 0;
    while (i < epochs_until_now) {
        // stop when we encounter epoch that is not final and reward is configured to wait
        for balance update.
        if (
             reward.balance_update_enabled && (
                 !reward.epoch_updates_finalized.contains(next_epoch_time) ||
                 !(*reward.epoch_updates_finalized.borrow(next_epoch_time))
             )
         ) {
             break
         };
```

Each loop iteration performs multiple table lookups, binary search operations, math operations and more, making each of them very expensive.

That is problematic, as when user is trying to claim a reward after long time, for example 2 years, this loop will be executed approx 104 times, most likely consuming all gas available.

2.7.2. Impact

Issue might lead to a Denial of Service scenario on a user trying to claim rewards.

2.7.3. Recommendation

We recommend implementation of countermeasures to prevent from it, allowing users to claim rewards anytime, no matter how long they were not doing it. This can by done through impleemntation of batched claiming, performing safe maximum claim `n` times up to the total `epochs_until_now`.

2.8. [M2] Missing voter `EmergencyCouncilCap` validation usage

Target	distribution/sources/voter.move
Severity	Medium
Category	Vulnerability

2.8.1. Description

The `distribution` module defines two validation-related functions for the `EmergencyCouncilCap` - `validate_emergency_council_voter_id` and `validate_emergency_council_minter_id`.

However, only `validate_emergency_council_minter_id` is ever invoked in the `kill`, `revive`, and `reset` gauge flows.

The `validate_emergency_council_voter_id` check is never used—so any `EmergencyCouncilCap` object, regardless of its voter field, can pass the voter-side check.

```
1  public fun validate_emergency_council_voter_id(emergency_council_cap:
   &EmergencyCouncilCap, voter_id: ID) {
2      assert!(emergency_council_cap.voter == voter_id, EEmergencyCouncilDoesNotMatchVoter);
3  }
```

It is worth to mention that the same scenario exists in the whitelisting logic, where calls to `validate_pair` are missing.

2.8.2. Impact

A cap minted for one governance instance or voter can be replayed against another. This breaks the assumption that only the correct emergency council voter can pause or unpause actions, opening an authorization bypass.

2.8.3. Recommendation

We recommend that before any critical council action, invoke both validations against the current `VotingEscrow` instance. Perform the same for the `validate_pair` function in the whitelisting logic as well.

2.9. [M3] Incorrect calculation of team emissions in the Minter contract

Target	distribution/sources/minter.move
Severity	Medium
Category	Vulnerability

2.9.1. Description

One of the functionality of `update_period` function is to calculate and distribute `SailCoin` to the team weekly. However, the `team_emissions` calculation is over estimated making the calculation wrong.

The `team_emissions` is calculated on top of normal weekly emissions and rebase growth in the `update_period` function as follows:

```
public fun update_period<SailCoinType, EpochOSail>(
    minter: &mut Minter<SailCoinType>,
    voter: &mut distribution::voter::Voter,
    distribution_config: &distribution::distribution_config::DistributionConfig,
    distribute_governor_cap: &DistributeGovernorCap,
    voting_escrow: &distribution::voting_escrow::VotingEscrow<SailCoinType>,
    reward_distributor: &mut
    distribution::reward_distributor::RewardDistributor<SailCoinType>,
    epoch_o_sail_treasury_cap: TreasuryCap<EpochOSail>,
    clock: &sui::clock::Clock,
     ctx: &mut TxContext
 ) {
     ...
     if (minter.team_emission_rate > 0 && minter.team_wallet != @0x0) {
         let team_emissions = integer_mate::full_math_u64::mul_div_floor(
             minter.team_emission_rate,
             rebase_growth + ending_epoch_emissions,
             RATE_DENOM - minter.team_emission_rate <----
         );
         transfer::public_transfer<Coin<SailCoinType>>(
             minter.mint_sail(team_emissions, ctx),
             minter.team_wallet
         );
     };
 }
```

The problem with the `team_emissions` calculation is that it divides by less than 100%, so effectively the team emissions will be more than expected. Let's say the `team_emission_rate` is the `MAX_TEAM_EMISSIONS_RATE` of `500`. `RATE_DENOM` is equal to `10000`, so the equation comes out as:

```
=> (500 * rebase_growth + ending_epoch_emissions) / (10000 - 500)
=> (rebase_growth + ending_epoch_emissions) * 500 / 9500
=> (rebase_growth + ending_epoch_emissions) * 0.0526315789
```

And instead of taking the maximum of 5% as the code clearly indicates, it'll take 5.26% instead.

2.9.2. Impact

Team emissions will be more than the maximum which will affect directly the issuance of SailCoin tokens and thus its inflation rate.

2.9.3. Recommendation

We recommend modifying the `Minter::update_period` function as shown below:

```
public fun update_period<SailCoinType, EpochOSail>(
    minter: &mut Minter<SailCoinType>,
    voter: &mut distribution::voter::Voter,
    distribution_config: &distribution::distribution_config::DistributionConfig,
    distribute_governor_cap: &DistributeGovernorCap,
    voting_escrow: &distribution::voting_escrow::VotingEscrow<SailCoinType>,
    reward_distributor: &mut
    distribution::reward_distributor::RewardDistributor<SailCoinType>,
    epoch_o_sail_treasury_cap: TreasuryCap<EpochOSail>,
    clock: &sui::clock::Clock,
     ctx: &mut TxContext
 ) {
     ...
     if (minter.team_emission_rate > 0 && minter.team_wallet != @0x0) {
         let team_emissions = integer_mate::full_math_u64::mul_div_floor(
             minter.team_emission_rate,
             rebase_growth + ending_epoch_emissions,
-            RATE_DENOM - minter.team_emission_rate
+            RATE_DENOM // Corrected to use RATE_DENOM directly
         );
         transfer::public_transfer<Coin<SailCoinType>>(
             minter.mint_sail(team_emissions, ctx),
             minter.team_wallet
         );
     };
 }
```

2.10. [M4] Pause flag not enforced on locked reward claims

Target	liquidity_locker/sources/liquidity_lock_v2.move
Severity	Medium
Category	Vulnerability

2.10.1. Description

In `liquidity_lock_v2.move`, while the staking style reward claim function (`claim_position_reward_for_staking`) checks the module's pause flag, the locked position reward entrypoints (`collect_reward` and `collect_reward_sail`) do not implement it.

If the admin pauses the contract to perform an emergency upgrade or halt operations, users could still claim rewards, altering state in unexpected ways during a paused period.

```
public fun claim_position_reward_for_staking<CoinTypeA, CoinTypeB, RewardCoinType>(
    locker: &Locker,
    gauge: &mut distribution::gauge::Gauge<CoinTypeA, CoinTypeB>,
    pool: &mut clmm_pool::pool::Pool<CoinTypeA, CoinTypeB>,
    locked_position: &LockedPosition<CoinTypeA, CoinTypeB>,
    clock: &sui::clock::Clock,
    ctx: &mut TxContext
) {
    checked_package_version(locker);
     assert!(!locker.pause, ELockManagerPaused);
```

2.10.2. Impact

Broken business logic related to the pausing mechanism.

2.10.3. Recommendation

We recommend adding the same pause check to every reward claim entrypoint:

```
checked_package_version(locker);
assert!(!locker.pause, ELockManagerPaused);
```

2.11. [M5] Gas Griefing Due to Incorrect Recursive Call in `get_prior_supply_index` Function

Target	distribution/sources/free_managed_reward.move
Severity	Medium
Category	Vulnerability

2.11.1. Description

The `get_prior_supply_index` function contains an implementation error where it recursively calls itself instead of calling the intended function on the internal reward object. The function is implemented as:

```
public fun get_prior_supply_index(reward: &FreeManagedReward, time: u64): u64 {
    reward.get_prior_supply_index(time)
}

public fun get_prior_balance_index(
    reward: &FreeManagedReward,
    lock_id: ID,
    time: u64
): u64 {
     reward.reward.get_prior_balance_index(lock_id, time)
 }
```

This creates an infinite recursion loop. The function should be calling `get_prior_supply_index` on the internal `reward.reward` object, similar to how other wrapper functions in this module are implemented (like `get_prior_balance_index`).

2.11.2. Impact

Any call to `get_prior_supply_index` will result in a stack overflow due to infinite recursion, causing the transaction to abort with `CALL_STACK_OVERFLOW`. This makes the function completely unusable. Additionally, this bug creates a gas griefing vector. Even though the transaction fails, it still consumes significant gas before hitting the stack limit.

2.11.3. Recommendation

We recommend fixing the function to properly delegate to the internal reward object:

```
  public fun get_prior_supply_index(reward: &FreeManagedReward, time: u64): u64 {
-     reward.get_prior_supply_index(time)
+     reward.reward.get_prior_supply_index(time)
  }
```

2.12. [M6] Desynchronization Between `Lock` Object Data and Internal `LockedBalance` State

Target	distribution/sources/voting_escrow.move
Severity	Medium
Category	Vulnerability

2.12.1. Description

The voting escrow system maintains two separate data representations: user-held `Lock` Object and internal `LockedBalance` state. Multiple functions fail to synchronize these data sources, creating inconsistencies:

- Merging - When merging locks, the resulting Lock Object retains outdated `end` timestamp while internal state uses the maximum end time of merged locks, enabling marketplace fraud
- Deposit managed - the `Lock` object retain original values (`amount` , `end` , `permanent`) after depositing into managed locks
- Withdraw managed - managed `Lock` object decrease by original deposited amounts while internal state accounts for accrued rewards

2.12.2. Impact

The issue enables marketplace fraud where attackers sell Lock object with misleading expiry dates and amounts. Additionally, any protocol integrating with `Lock` object (lending, AMMs, bridges) will make decisions based on stale/incorrect data, risking cascading failures.

2.12.3. Recommendation

We recommend adding synchronization logic to all state-modifying functions to update `Lock` object fields:

```
  public fun merge<SailCoinType>(...)
  {
      // ... existing merge logic ...
      lock_b.amount = new_locked_balance.amount;
+     lock_b.end = new_locked_balance.end;
  }

  public fun deposit_managed<SailCoinType>(...)
  {
       // ... existing deposit_managed logic ...
       managed_lock.amount = managed_lock.amount + current_locked_amount;
+     lock.amount = 0;
+     lock.end = 0;
+     lock.permanent = true;
   }

   public fun withdraw_managed<SailCoinType>(...)
   {
       // ... existing withdraw_managed logic ...
       lock.amount = new_managed_weight;
       lock.permanent = false;
       lock.end = lock_end_time;
-      managed_lock.amount = managed_lock.amount - managed_weight;
       // ...
       let mut managed_lock_balance = *voting_escrow.locked.borrow(managed_lock_id);
       let mut remaining_amount = if (new_managed_weight < managed_lock_balance.amount) {
           managed_lock_balance.amount - new_managed_weight
       } else {
           0
       };
       managed_lock_balance.amount = remaining_amount;
+      managed_lock.amount = remaining_amount;
   }
```

2.13. [M7] Missing function responsible for `team_cap` creation

Target	distribution/sources/voting_escrow.move
Severity	Medium
Category	Vulnerability

2.13.1. Description

In the `voting_escrow` the `toggle_split` can be called by user with `team_cap`, however in the `team_cap.move`, the `create` function is not used anywhere in the codebase. So in fact, as it is `(package)` one, it cannot be called by external caller, and `team_cap` cannot be created, making the `toggle_split` function unusable.

```
public fun toggle_split<SailCoinType>(
    voting_escrow: &mut VotingEscrow<SailCoinType>,
    team_cap: &distribution::team_cap::TeamCap,
    who: address,
    allowed: bool
) {
    team_cap.validate(object::id<VotingEscrow<SailCoinType>>(voting_escrow));
    if (voting_escrow.can_split.contains(who)) {
        voting_escrow.can_split.remove(who);
     };
     voting_escrow.can_split.add(who, allowed);
     let toggle_split_event = EventToggleSplit {
         who,
         allowed,
     };
     sui::event::emit<EventToggleSplit>(toggle_split_event);
 }
```

```
public(package) fun create(target: ID, ctx: &mut TxContext): TeamCap {
    TeamCap {
        id: object::new(ctx),
        target,
    }
}

public(package) fun validate(team_cap: &TeamCap, arg1: ID) {
    assert!(team_cap.target == arg1, ETeamCapInvalid);
 }
```

2.13.2. Impact

Without `team_cap`, nobody can call the `toggle_split` function.

2.13.3. Recommendation

We recommend adding a way for `team_cap` creation, based on current logic of the solution.

2.14. [M8] Incomplete state cleanup when killing gauges

Target	distribution/sources/minter.move
Severity	Medium
Category	Vulnerability

2.14.1. Description

The `kill_gauge` function only updates the gauge's liveness status without handling positions that were locked via the protocol-controlled `lock_position` mechanism.

Since `LockerCaps` are exclusively controlled by the protocol (only `CreateCap` holders can mint them), and the `kill_gauge` function doesn't trigger any unlock operations, positions in the `locked_positions` table become permanently inaccessible when a gauge is killed.

Additionally, the function leaves orphaned state in the voter module:

- Historical votes in `voter.votes` table
- Gauge weights in `voter.weights` table

While these don't affect emission calculations, as emissions are gauge-specific, they represent permanent state bloat.

```
public fun kill_gauge<SailCoinType>(
    minter: &mut Minter<SailCoinType>,
    distribution_config: &mut distribution::distribution_config::DistributionConfig,
    emergency_council_cap: &distribution::emergency_council::EmergencyCouncilCap,
    gauge_id: ID,
) {
    emergency_council_cap.validate_emergency_council_minter_id(object::id(minter));
    assert!(
        minter.is_valid_distribution_config(distribution_config),
         EKillGaugeDistributionConfigInvalid
     );
     assert!(
         distribution_config.is_gauge_alive(gauge_id),
         EKillGaugeAlreadyKilled
     );
     distribution_config.update_gauge_liveness(vector<ID>[gauge_id], false);
     let kill_gauge_event = EventKillGauge { id: gauge_id };
     sui::event::emit<EventKillGauge>(kill_gauge_event);
 }
```

2.14.2. Impact

Any positions in `gauge.locked_positions` become inaccessible since only protocol-controlled `LockerCaps` can unlock them, and there's no emergency unlock mechanism.

2.14.3. Recommendation

We recommend implementing a cleanup function to handle gauge termination properly.

2.15. [L1] Missing global event emission convention

Target	distribution/*
Severity	Low
Category	Vulnerability

2.15.1. Description

In the `distribution` module, the event emission is not standarized for state-changing operations. For some of them, emission is performed.

However for others, like `set_team_emission_rate` or `set_protocol_fee_rate`, emission is missing.

```
public fun set_team_emission_rate<SailCoinType>(
    minter: &mut Minter<SailCoinType>,
    admin_cap: &AdminCap,
    team_emission_rate: u64
) {
    minter.check_admin(admin_cap);
    assert!(!minter.is_paused(), ESetTeamEmissionRateMinterPaused);
    assert!(team_emission_rate <= MAX_TEAM_EMISSIONS_RATE, ESetTeamEmissionRateTooBigRate);
    minter.team_emission_rate = team_emission_rate;
 }
```

That might lead to a problematic offchain monitoring, as aggregators will not be able to react quickly for critical system changes.

2.15.2. Impact

Missing easy ability to track all system critical operations offchain.

2.15.3. Recommendation

We recommend standarizing the event emission strategy ensuring, that all critical system operations returns proper event to be tracked offchain.

2.16. [L2] Uninitialized Emergency Council with No Update Mechanism

Target	distribution/sources/voter.move
Severity	Low
Category	Vulnerability

2.16.1. Description

The Voter contract initializes the `emergency_council` field to a zero address (`@0x0`) during creation and provides no function to update it. While this field is currently unused, its presence suggests planned emergency governance functionality that cannot be implemented due to the permanently invalid address.

2.16.2. Impact

If future updates introduce functions dependent on `emergency_council` for authorization or emergency procedures, they will fail due to the invalid address. This could prevent critical emergency responses during security incidents or protocol vulnerabilities.

2.16.3. Recommendation

We recommend adding an access-controlled setter function to configure the emergency council address post-deployment. Include proper authorization checks and event emission for transparency. Alternatively, if emergency functionality is not planned, remove the unused field.

2.17. [L3] Unnecessary Lock Duration Validation for Permanent Locks Causes User Friction

Target	distribution/sources/voting_escrow.move
Severity	Low
Category	Vulnerability

2.17.1. Description

The `create_lock` function allows users to lock tokens either for a fixed number of days or permanently using the `permanent` flag. However, even when permanent is set to `true`, the function still checks if `lock_duration_days` is within an allowed range using `validate_lock_duration`. This check doesn't make sense for permanent locks since the lock duration is not used. If a user provides a value outside the allowed range—thinking it doesn't matter because they are making a permanent lock-the transaction will revert, causing confusion and a bad experience.

Additionally, there is an inconsistency in the data. The `end_time` is not updated in the user's `Lock` object, even though it gets overridden to 0 for permanent locks in the internal logic. This means the user's stored object might show an end time even though the lock is meant to be permanent, which can be misleading.

2.17.2. Impact

Users trying to create permanent locks might see their transaction fail due to an unnecessary duration check. It can also create confusion as their Lock object may show a non-zero expiry, even though the lock is permanent.

2.17.3. Recommendation

We recommend skipping the `validate_lock_duration` check when permanent is true. Also update `end_time` in Lock object aliging to internal storage.

2.18. [L4] Inconsistent and Unused Reward Claim Logic Across Voting Reward Modules

Target	distribution/sources/bribe_voting_reward.move
Severity	Low
Category	Vulnerability

2.18.1. Description

The `voter_get_reward` function is defined in three modules — `exercise_fee_reward`, `fee_voting_reward`, and `bribe_voting_reward`. These functions are intended to allow a voter contract to claim rewards for a specific lock by validating an authorization capability. However, the validation mechanism is inconsistent across the modules:

- In `exercise_fee_reward` and `fee_voting_reward`, the function validates a `VoterCap`, which directly proves that the caller is the voter contract.

In contrast, the `bribe_voting_reward` module uses a `RewardAuthorizedCap` instead of `VoterCap`. This makes it unclear who is authorized to call `voter_get_reward` in this context. Since it doesn't validate that the caller is the original voter, it could potentially allow unauthorized reward claims if `RewardAuthorizedCap` is misused.

Moreover, although all three modules define the `voter_get_reward` function, none of them are actually used anywhere in the current codebase. These functions are presumably meant to be called by a voter contract, but that integration is missing, leaving these functions unused and possibly untested.

2.18.2. Impact

The inconsistency in access control (use of `RewardAuthorizedCap` instead of `VoterCap`) in `bribe_voting_reward` could lead to confusion or even authorization flaws if misused.

Additionally, if these functions are mistakenly assumed to be in use, rewards may not be claimable by voters as intended from voter contract.

2.18.3. Recommendation

We recommend standarizing the access control mechanism by using VoterCap consistently across all three reward modules, or clearly document and justify the difference in the `bribe_voting_reward` module.

We suggest either integrating these `voter_get_reward` functions properly into the voter contract or remove them if they are no longer relevant.

2.19. [I1] Gas inefficiency in fixed point price conversions

Target	liquidity_locker/sources/locker_utils.move
Severity	Informational
Category	Gas

2.19.1. Description

In `calculate_position_liquidity_in_token_a`, to convert amounts between token A and token B using a Q64.64 price, the code currently performs three consecutive 64 bit left shifts on a u256 value:

```
// Convert balance_b to tokenA equivalent
let amount_b_in_a = ((((amount_b as u256) << 64) << 64) << 64) / price; // Q64.64
```

Each shift on a 256-bit integer is expensive in gas. A single shift by 192 bits (`<< 192`) would achieve the same result.

2.19.2. Impact

Such operation is ineficcient.

2.19.3. Recommendation

We recommend checking the shifting logic accordingly.

2.20. [I2] Incorrect Abort Documentation for Locked Token Claims

Target	distribution/sources/reward_distributor.move
Severity	Informational
Category	Vulnerability

2.20.1. Description

The `claim()` function in the reward distributor contains documentation that directly contradicts its actual implementation. The function comment states `If the voting escrow is not locked` as an abort condition, but the code implements the exact opposite logic.

```
1 voting_escrow.escrow_type(lock_id).is_locked() == false
```

However the escrow must NOT be locked (i.e., must be NORMAL or MANAGED type) to claim rewards.

2.20.2. Impact

This documentation error can lead to significant user confusion and incorrect integration implementations.

2.20.3. Recommendation

We recommend updating the documentation to accurately reflect the code behavior.

2.21. [I3] Mutable object used in `get_lock_periods` getter

Target	liquidity_locker/sources/liquidity_lock_v2.move
Severity	Informational
Category	Vulnerability

2.21.1. Description

The `get_lock_periods` uses the `locker` parameter of `Locker` type, marking it as `&mut` , while the whole operation is a public getter. This is incorrect, as getters are treated as read-only operations, and cannot mutate the state variables by design.

```
public fun get_lock_periods(
    locker: &mut Locker,
): (vector<u64>, vector<u64>) {
    (locker.periods_blocking, locker.periods_post_lockdown)
}
```

2.21.2. Impact

If the function logic will be changed in the future, this can lead to serious state inconsistencies.

2.21.3. Recommendation

We recommend removing the `&mut` operator from `locker` parameter.

2.22. [I4] The `reserve_split` does not check if amount is smaller than reserves balance

Target	distribution/sources/gauge.move
Severity	Informational
Category	Vulnerability

2.22.1. Description

In the `distribution` module `gauge.move`, the `reserves_split` performs such an operation:

```
gauge.reserves_all_tokens = gauge.reserves_all_tokens - amount;
```

Without checking, if `gauge.reserves_all_tokens >= amount`.

```
fun reserves_split<CoinTypeA, CoinTypeB, RewardCoinType>(
    gauge: &mut Gauge<CoinTypeA, CoinTypeB>,
    amount: u64,
): Balance<RewardCoinType> {
    // keep behaviour of builtin split func, so it will not fail with zero amount even if there are no reserves
    if (amount == 0) {
        return balance::zero<RewardCoinType>()
    };
    let coin_type = type_name::get<RewardCoinType>();

     gauge.reserves_all_tokens = gauge.reserves_all_tokens - amount;

     gauge
         .reserves_balance
         .borrow_mut<TypeName, Balance<RewardCoinType>>(coin_type)
         .split(amount)
 }
```

If `amount` exceeds `gauge.reserves_all_tokens`, the subtraction underflows and aborts the transaction with a generic arithmetic-error code.

2.22.2. Impact

Caller will see a low-level abort rather than a semantic “insufficient reserves” error, making root cause debugging harder.

2.22.3. Recommendation

We recommend adding an explicit precondition check with a clear error code before performing the subtraction.

2.23. [I5] Missing Updates to `last_epoch_update_time` Field

Target	distribution/sources/minter.move
Severity	Informational
Category	Vulnerability

2.23.1. Description

The `Minter::last_epoch_update_time` field is broken. It gets set once when the minter starts but never gets updated when epochs actually change.

The code has a function called `last_epoch_update_time` that suggests it tracks when epochs were last updated, but it always returns the activation time instead. This happens because the `update_period_internal` function forgets to update this field even though it should.

2.23.2. Impact

It also breaks any external systems trying to monitor if the protocol is working properly since they get old timestamp data. The misleading function name will confuse other developers who expect it to work as advertised.

2.23.3. Recommendation

We recommend fixing this by adding one line to update the timestamp in the `update_period_internal` function.