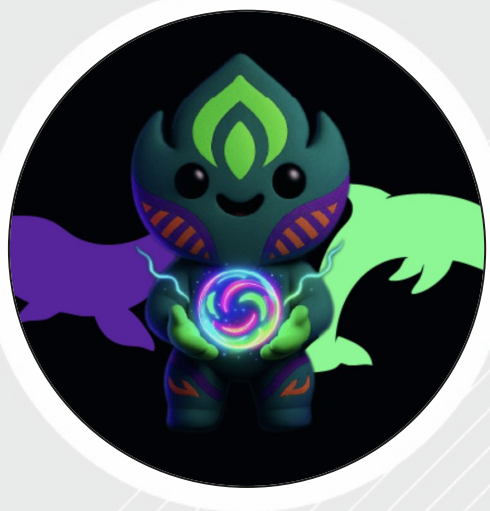
SPYWOLF

Security Audit Report



Audit prepared for

Lotus

Completed on

September 1, 2025

TABLE OF CONTENTS

PROJECT INFORMATION

Project & Scope

- **Name**: Lotus
- **Description**: OTC Flat-Price Launchpad (LOTUSV1)
- **Platform**: Sui (Move)
- **Primary Module**: lotus::LOTUSV1
- **Audit Snapshot**: Source provided by client; public claims verified on Aug 18, 2025.

Lifecycle in brief (as marketed)

- **Phase 1 - Lotus Round**: A total of 100 blocks are for sale, with a limit of one block per wallet. Each block contains 4M coins. The first 75 blocks constitute the buyback phase, during which buyers can sell back their block once for a full refund. After 75 blocks are sold, the sellback option is disabled and a 24-hour countdown begins.
- **Phase 2 - Rewardian Round**: The final 25 blocks must be sold before the 24-hour timer expires for the project to "graduate" and unlock community rewards. If the sale does not sell out in time, the rewards are burned.
- **Phase 3 - SRM**: Upon a successful graduation, 100% of the SUI raised is used to create a liquidity pool on the SRM Dex, paired with 30% of the total coin supply. This LP is advertised as being "permanently locked" within the SRM Dex.

Stated Distribution & Rewards (as marketed))

- **Supply Splits on Sell-out**: 40% of the total supply is allocated for the sale, 30% for the bonus rewards pool, and 30% for the liquidity pool. The 30% reward pool is split among the community as follows:
  - 2 Whale airdrops of 2% each.
  - 8 Dolphin airdrops of 1% each.
  - All other holders receive a 1.5x bonus on their holdings.
  - The developer wallet is excluded from eligibility for the Whale and Dolphin airdrops.

Fair-launch principles

- The project emphasizes fair-launch principles such as "No bonding curves," "No early whales," and the "same price for everyone". The "one block = one shot" rule is enforced on-chain, though it is acknowledged that this does not prevent users from participating with multiple wallets.

Roles & Promised Mechanics (public)

- **Buyers**: Participants are limited to one block purchase per wallet and may sell it back once before the 75-block threshold is met.
- **Graduation Triggers**: The launch is considered guaranteed and the 24-hour timer starts once 75 blocks are sold. Community rewards are unlocked only if all 100 blocks are sold.
- **Liquidity**: All raised SUI is claimed to be paired with 30% of the total token supply to form the liquidity pool, which is then permanently locked on the SRM Dex.

SCOPE OF AUDIT(1)

Objectives

- **Primary goal:** Assess whether the on-chain Move module (lotus::LOTUSV1) safely implements the publicly advertised Lotus Launch mechanics and preserves user funds under adversarial conditions.
- **Secondary goal:** Identify any divergence between **marketed promises** (buyback guarantees, LP "permanent lock," rewards behavior) and **what the code actually enforces**, and evaluate the associated risk.

Codebase & Artifacts Reviewed

- **Files:** lotus.move, Move.toml (snapshot received Aug 22, 2025, Europe/Budapest).
- **Primary module:** module lotus::LOTUSV1.
- **Core types:** Pool<phantom A, phantom B>, Factory, Config, CreatePoolLock.
- **Entry points examined:** Pool creation (new_lotus_pool, create_lotus_pool), trading (buy_block/sell_block → trade_a_for_b/trade_b_for_a), lifecycle toggles (set_pool_closed, set_pool_buyback_closed), withdrawals (withdraw_*), and config updates (update_*).
- **Generics:** All paths checked under generic coin types **A** (payment) and **B** (launch token).

Note 1: Review is **source-based**. No off-chain servers, frontends, or SRM Dex contracts were included in this code snapshot.
Note 2: While the SRM Dex contracts are out of scope for this specific audit, they were subject to a separate, comprehensive security audit by our team, which verified the functionality of its permanent liquidity locking mechanism.

Attack Surfaces Considered (In Scope)

1. **Access Control & Roles:** Admin vs. Launch Manager authorities; ability to pause/close buyback, withdraw reserves, update fees/wallets.
2. **Funds Flow & Accounting:** A and B reserves, buy/sell math, fee accrual & distribution, invariant preservation, and rounding/overflow checks.
3. **Lifecycle Safety:** Pool creation invariants; buyback open/close conditions; 24h deadline semantics; sold-out handling; finalization and post-launch withdrawals.
4. **User Protections:** "One-purchase-per-wallet" enforcement; sellback eligibility; refund and solvency assumptions; denial-of-service on trading.
5. **Economic/Protocol Risks:** Centralization levers (manager/admin); premature liquidity or reward extraction; insolvency of buyback reserves; griefing vectors.
6. **Inter-object & Registry Safety:** Factory and pool registration integrity; uniqueness by (A,B); allowlist gating in CreatePoolLock.
7. **Telemetry & Observability:** Event coverage and correctness sufficient for indexers/monitors to reconstruct state transitions.

SCOPE OF AUDIT(2)

Explicit Exclusions (Out of Scope)

- **Off-chain infrastructure:** webforms, server hot-wallet ops, deployment pipelines.
- **External protocols: SRM Dex** contracts (liquidity creation/locking), any bridges, oracles, or randomness beacons.
- **Frontends/SDKs:** UI logic, client signing flows.
- **Economic guarantees not encoded on-chain:** marketing claims that rely purely on operator honesty unless explicitly enforced by this module.

Assumptions & Constraints

- **Time source:** Contract uses on-chain timestamp where applicable; availability and monotonicity assumed per Sui guarantees.
- **Token behavior:** Standard Sui Coin<T> semantics; no custom decimals/fee-on-transfer tokens that alter expected math.
- **Operator behavior:** Where the code delegates actions (e.g., migration to SRM, “lock LP”), only **on-chain checks** are treated as enforceable guarantees.
- **Single-block sellback:** Design intends sellback of **exactly one full block** unit; partial sells considered out of scope unless code permits them.

Methodology

- Manual line-by-line review of entry points and internal helpers.
- Property-driven reasoning about invariants (reserves ≥ 0, buyback solvency vs. thresholds, one-per-wallet).
- Role/permission diffs between **intended** and **enforced** behavior.
- Event/state congruence checks (does each critical state flip emit an unambiguous event?).
- Adversarial thought experiments (rug/DoS via withdrawals, early buyback closure, misconfigured pool creation).

Deliverables From This Audit

- **Findings catalogue:** Ranked **Critical/High/Medium/Low/Info** with impact, exploit scenarios, and concrete code-level remediations.
- **Design gaps:** Clear mapping of **promised** vs **enforced** features (e.g., LP lock, rewards rules).
- **Suggested patches:** Minimal, actionable guard conditions and lifecycle changes to harden user protections.
- **Test plan:** Cases to reproduce bugs and to prevent regressions (buyback solvency, deadline enforcement, withdrawal gating, event assertions).

FINDINGS

🟥 Critical Risk

✅ RESOLVED

C-01: Unrestricted Reserve Withdrawals Enable Rug/Buyback DoS

Description:
All reserve withdrawal functions allow the **launch manager** to drain funds **at any time**, regardless of pool state (sale in progress, buyback open, or before the 24h deadline). Withdrawing **Coin A** breaks buyback solvency immediately; withdrawing **LP deposit** or **rewards** contradicts the promise of "locked liquidity" and conditional rewards.

Code (unchanged, from lotus.move L240–L253):

```
public entry fun withdraw_balance_a<A, B>(
    pool: &mut Pool<A, B>,
    config: &Config,
    ctx: &mut TxContext
) {
    let caller = sender(ctx);
    assert!(caller == config.launch_manager, EUnauthorized);

    let available_balance_a = balance::value(&pool.balance_a);

    let balance_a = balance::split(&mut pool.balance_a, available_balance_a);

    destroy_zero_or_transfer_balance(balance_a, config.launch_manager, ctx);
}
```

Recommendation:

Gate **all** withdrawals behind a **finalized terminal state** and explicit **deadline** checks. At minimum:

- Require pool.buyback_open == false **and** (pool.sold_out == true || now >= close_deadline) **and** pool.pool_closed == true.
  Add a single, permissioned-but-verifiable finalize() that (a) enforces the 24h rule, (b) creates/locks LP on-chain or hands off to a **locking contract**, (c) atomically distributes/burns rewards per outcome, and (d) only after that exposes narrowly scoped withdrawals.
- Emit unambiguous events on finalization and withdrawals.

Implemented Fixes:

All withdrawal functions (withdraw_balance_a, withdraw_balance_b, withdraw_rewards, and withdraw_lp_deposit) are now gated by a new internal function called assert_can_withdraw. This function strictly requires that the pool is in a terminal state (either pool.sold_out == true or the close_deadline has passed) before any assets can be moved. This completely prevents the launch manager from draining funds while the sale or buyback period is active.

FINDINGS

🟥 Critical Risk ✅ RESOLVED

C-02: Launch Manager Can Unilaterally Close Buyback Early

Description:
set_pool_buyback_closed lets the **launch manager** close buyback **whenever they want** and start the 24h countdown, even if the configured buyback threshold hasn't been reached. Together with C-01, this creates a centralized kill-switch that can trap users (cannot sell back) and then allow reserves to be withdrawn.

Code (unchanged, L337–L351):

```
public entry fun set_pool_buyback_closed<A, B>(
    pool: &mut Pool<A, B>,
    config: &Config,
    clock: &Clock,
    ctx: &mut TxContext
) {
    let caller = sender(ctx);
    assert!(caller == config.launch_manager, EUnauthorized);

    let now = get_timestamp(clock);
    let deadline = now + 86_400_000; // 24h in ms

    pool.buyback_open = false;
    pool.close_deadline = option::some(deadline);
```

Recommendation:

Make buyback closure **automatic and objective**:

- Remove (or restrict) this entry function; instead, close buyback **inside buy path** when blocks_sold >= buyback_limit.
- If the function must remain, enforce:
  - assert!(pool.buyback_open, EBuybackClosed);
  - assert!(pool.blocks_sold >= pool.buyback_limit, EBuybackClosed);
- Add a **permissionless** expire() callable by anyone after close_deadline to flip pool.pool_closed = true (prevents operator abuse).

Implemented Fixes:

The manual set_pool_buyback_closed entry function was removed. Instead, the buyback window closure is now automated within the trade_a_for_b function. It trustlessly triggers only when the number of blocks sold reaches the buyback_limit. This removes the centralized risk and ensures the buyback window rules are enforced by the contract itself.

FINDINGS

🟥 Critical Risk ✅ RESOLVED

C-03: LP Deposit & Rewards Are Not Locked On-Chain (Withdrawable Anytime)

Description:

Despite marketing claims of **permanently locked LP** and **conditional rewards**, both the **LP deposit** and **reward pool** are withdrawable by the launch manager at any time, with no "sold out" or deadline gating. This is a direct contradiction to promised lockups and exposes users to post-raise misappropriation.

Code (unchanged, L302–L315 and L287–L300):

```
public entry fun withdraw_lp_deposit<A, B>(
    pool: &mut Pool<A, B>,
    config: &Config,
    ctx: &mut TxContext
) {
    let caller = sender(ctx);
    assert!(caller == config.launch_manager, EUnauthorized);

    let available_lp = balance::value(&pool.lp_deposit);

    let lp_deposit = balance::split(&mut pool.lp_deposit, available_lp);

    destroy_zero_or_transfer_balance(lp_deposit, config.launch_manager, ctx);
}
```

```
public entry fun withdraw_rewards<A, B>(
    pool: &mut Pool<A, B>,
    config: &Config,
    ctx: &mut TxContext
) {
    let caller = sender(ctx);
    assert!(caller == config.launch_manager, EUnauthorized);
```

Recommendation:

- Encode a **formal finalization flow** that:
  a. Locks LP via an **on-chain lock contract** (or irrevocably deposits LP tokens into a non-transferable object).
  b. Distributes rewards only when `sold_out == true` before the deadline; otherwise burns to a verifiable sink.
- Until finalization completes, **forbid** `withdraw_lp_deposit` and `withdraw_rewards`.
- If off-chain SRM migration is unavoidable, create a **timelocked escrow** object with unambiguous, auditable release conditions and a permissionless fail-safe to burn/return funds on timeout.

Implemented Fixes:

This is resolved by the same fix as C-01. The addition of the `assert_can_withdraw` check to the `withdraw_lp_deposit` and `withdraw_rewards` functions ensures that these funds cannot be touched until the sale has concluded, aligning with the project's lifecycle promises.

FINDINGS

High Risk

H-01: Sellback is not limited to original buyers (and not “once per wallet”)

Description:
During buyback, **any holder** of exactly one block of B can redeem for A—there’s **no check** that the caller ever bought from the pool, and there’s **no per-wallet sell limit**. This enables aggregators/whales to acquire many blocks OTC and dump them to the contract during buyback, rapidly draining A reserves and undermining the “buyers may sell back once” promise and overall fairness.

Code (unchanged, from trade_b_for_a):

```
assert!(!pool.pool_closed, EPoolClosed);
assert!(pool.buyback_open, EBuybackClosed);
assert!(!pool.sold_out, EPoolSoldOut);

// 2. Validate input amount is exactly 1 block
let input_amount = balance::value(&input);
assert!(input_amount == pool.block_quantity, EInvalidTradeAmount);
```

Recommendation:

Enforce sellback eligibility and limits:

- Require the caller to be a recorded buyer: assert!(table::contains(&pool.buyers, sender_addr), EUnauthorized);
- Track and enforce **one sellback per wallet** (e.g., a sellers map, or flip a per-wallet “redeemed” flag).
- Optionally, track **total sellbacks** to guard against mass OTC aggregation attacks (e.g., cap sellbacks at buyback_limit blocks or maintain a running redemption counter).

FINDINGS

🟥 High Risk ✅ RESOLVED

H-02: 24h deadline is not enforced in buy/sell paths (operator-timed window)

Description:

The module records a close_deadline when buyback closes, but **neither** buy_block nor sell_block (nor the internal trade functions) check now <= close_deadline. Trading is only gated by pool_closed, which the **launch manager** can toggle at will. This contradicts the "24h" guarantee and enables sales/sellbacks outside the intended window.

Code (unchanged, from sell_block):

```
public entry fun sell_block<A, B>(
        pool: &mut Pool<A, B>,
        config: &Config,
        mut input: Coin<B>,
        clock: &Clock,
        ctx: &mut TxContext
    ) {
        let required_input = pool.block_quantity;
        let input_value = coin::value(&input);

        assert!(input_value >= required_input, EInsufficientInput);

        let sender = sender(ctx);

        if (input_value == required_input) {
            let a_out = trade_b_for_a(pool, config, coin::into_balance(input), clock, ctx);
            destroy_zero_or_transfer_balance(a_out, sender, ctx);
        } else {
            let used_b = coin::split(&mut input, required_input, ctx);
            let a_out = trade_b_for_a(pool, config, coin::into_balance(used_b), clock, ctx);
            destroy_zero_or_transfer_balance(coin::into_balance(input), sender, ctx);
            destroy_zero_or_transfer_balance(a_out, sender, ctx);
        }
    }
```

Recommendation:

Make the deadline **objective and enforceable on-chain**:

- In buy_block/sell_block (or trade_*), add:
  if (option::is_some(&pool.close_deadline)) assert!(now <= *option::borrow(&pool.close_deadline), EPoolClosed);
- Add a **permissionless** expire() callable after the deadline to set pool_closed = true (so anyone can finalize timeouts).
- Disallow set_pool_closed(false) after a deadline has passed (see H-03).

Implemented Fixes:

The buy_block function now includes a check at the beginning to verify that the current timestamp has not passed the close_deadline. This ensures that once the 24-hour countdown begins, no further purchases can be made after it expires.

FINDINGS

🟥 High Risk ✅ RESOLVED

H-03: Manager can freely toggle pool open/closed at any time

Description:
set_pool_closed lets the launch manager set pool.pool_closed to **any** status at any time, without sold-out/deadline checks. They can arbitrarily **freeze** trading mid-sale (DoS) or **reopen** trading after the countdown, undermining the advertised timeline and user expectations.

Code (unchanged, from set_pool_closed):

```
assert!(caller == config.launch_manager, EUnauthorized);

pool.pool_closed = status;

let now = get_timestamp(clock);

event::emit(PoolClosed {
    pool_id: object::id(pool),
    timestamp: now
});
```

Recommendation:

Constrain state transitions:

- Forbid reopening after either `(sold_out == true)` or `now > close_deadline` (if set).
- Only allow closing when (a) buyback threshold reached (or sold out), or (b) deadline exceeded.
- Emit events that include the **new status** and the **reason** (deadline, sold-out, admin pause), so indexers can audit intent.

Implemented Fixes:

The set_pool_closed function is now also protected by the assert_can_withdraw check, meaning it can only be called after the sale has ended. Additionally, a new check was added to prevent the pool from being reopened (status = false) once it has reached a terminal state.

FINDINGS

🟥 High Risk ✅ RESOLVED

H-04: Deposit ratio promises (40/30/30) are not enforced; LP share can drift

Description:
Pool creation in new_lotus_pool first earmarks **exact sale supply** (block_quantity * max_blocks), then takes **30% of total** for rewards, and leaves the **remainder** as LP. There's **no assertion** that the provided amount_b actually equals **100%** of the intended supply (i.e., sale = 40%, rewards = 30%, LP = 30%). Misconfigured totals will silently change the LP share from the promised 30%.

Code (unchanged, from new_lotus_pool):

```
// 40% deposit
let init_b_deposit = coin::split(&mut used_b, expected_amount_b, ctx);
assert!(coin::value(&init_b_deposit) == expected_amount_b, EInvalidBalanceB);

// 30% rewards
let thirty_percent = (total_b * 3) / 10;
assert!(thirty_percent > 0, EInvalidSmallAmount);

let init_reward_b = coin::split(&mut used_b, thirty_percent, ctx);

// Remaining 30% in used_b
let init_lp_b = used_b;
```

Recommendation:

Enforce the advertised split:

- Assert total_b == expected_amount_b * 10 / 4 (i.e., **2.5** × sale supply), then derive rewards and LP strictly from total_b.
- Alternatively, take **all** of amount_b and compute:
  sale = total_b * 4 / 10, rewards = total_b * 3 / 10, lp = total_b - sale - rewards; then **assert** divisibility and exactness to avoid rounding drift.
- Emit the computed ratios in LotusPoolCreated to allow external verification.

Implemented Fixes:

The new_lotus_pool function was significantly updated to enforce this ratio. It now calculates the required total B supply based on a 2.5x multiple of the sale supply (which corresponds to 40%) and asserts that the input

amount_b matches this total. It then derives the sale (40%), rewards (30%), and LP (30%) portions from this total, with assertions to prevent rounding drift.

FINDINGS

Medium Risk

M-01: Post-deploy config mutability (admin/manager/wallet/fee) with no “freeze” control

Description:
Admin can change the global trade fee, fee wallet, admin, and launch manager **at any time**, including mid-sale. While the fee is capped, this still permits material policy changes during a launch and undermines “immutable after launch” expectations.

Code (unchanged):

```
public entry fun update_trade_fee(config: &mut Config, new_fee: u64, ctx: &mut TxContext) {
    let caller = sender(ctx);
    assert!(caller == config.admin, EUnauthorized);
    assert!(new_fee <= MAX_TRADE_FEE, EInvalidFee);
    config.trade_fee = new_fee;
}
```

```
public entry fun update_trade_fee_wallet(config: &mut Config, new_wallet: address, ctx: &mut TxContext) {
    let caller = sender(ctx);
    assert!(caller == config.admin, EUnauthorized);
    config.trade_fee_wallet = new_wallet;
}
```

Recommendation:

- Introduce a **one-way freeze_config()** (admin-only) that, once called, permanently disables update_* entries for the duration of active pools (or globally). Alternatively, snapshot the relevant parameters into each Pool at creation time (see M-05) so mid-sale changes to Config don’t affect existing launches.

FINDINGS

Medium Risk ✅ RESOLVED

M-02: PoolClosed event omits new status and reason (weak observability)

Description:
When toggling the pool open/closed, the emitted PoolClosed event does **not** include whether the pool is now open or closed, nor the cause (deadline, sold-out, admin pause). This makes off-chain monitoring and dispute resolution harder and complicates compliance with the advertised 24h flow.

Code (unchanged):

```
public struct PoolClosed has copy, drop {
    pool_id: ID,
    timestamp: u64
}
```

```
event::emit(PoolClosed {
    pool_id: object::id(pool),
    timestamp: now
});
```

Recommendation:

- Extend the event with status: bool and perhaps an enum-like reason ({deadline, sold_out, admin_pause}). Only emit on **state change**. Indexers and users should be able to reconstruct state transitions from events alone.

Implemented Fixes:

The PoolClosed event struct has been updated to include status: bool and reason: vector<u8>. The set_pool_closed function now correctly populates these fields, emitting whether the pool was closed (status) and why (b"sold_out" or b"deadline_passed") . This vastly improves off-chain observability.

FINDINGS

Medium Risk ✅ RESOLVED

M-03: Address updates accept 0x0 (blackhole / lockout risk)

Description:
Several admin update functions do not validate the new address. Setting the trade-fee wallet to @0x0 would **blackhole fees**; setting admin or launch_manager to @0x0 could unintentionally **brick** governance or operations. Notably, pool creation **does** validate creator_royalty_wallet != @0x0, demonstrating the pattern is known but inconsistently applied.

Code (unchanged):

```
public entry fun update_admin(config: &mut Config, new_admin: address, ctx: &mut TxContext) {
    let caller = sender(ctx);
    assert!(caller == config.admin, EUnauthorized);
    config.admin = new_admin;
}
```

```
assert!(creator_royalty_wallet != @0x0, EInvalidAddress);
```

Recommendation:

- Add assert!(new_* != @0x0, EInvalidAddress) to update_admin, update_launch_manager, and update_trade_fee_wallet. If the intent is to allow self-bricking for immutability, expose a **dedicated** renounce_*() path with clear documentation and events.

Implemented Fixes:

The update_trade_fee_wallet, update_admin, and update_launch_manager functions now all include an assert!(new_address != @0x0, EInvalidAddress) check, preventing the accidental bricking of controls.

FINDINGS

Medium Risk

M-04: Trade fee is read from global Config at execution time (not snapshotted per pool)

Description:
Both buy and sell use config.trade_fee **at the moment of trade**, meaning an admin fee change mid-sale alters costs retroactively for remaining participants. This can surprise users and break “fixed-fee at launch” expectations even if the cap is low.

Code (unchanged):

```
let (fee, expected_input) = compute_total_input(pool.block_price, config.trade_fee);
```

```
let (fee, _) = compute_total_input(block_price, config.trade_fee);
```

Recommendation:

- Snapshot trade_fee and destination wallet into the Pool at creation (e.g., pool.trade_fee_bps, pool.fee_wallet) and use those in trade_*. Optionally allow a **pre-launch-only** update window before the first block is sold, then freeze.

FINDINGS

Medium Risk ✅ RESOLVED

M-05: Allowlist is a vector with linear scans and no duplicate checks (operational DoS risk)

Description:

Pool creation allowlist uses vector::contains/index_of, which are **O(n)**. Large allowlists increase gas and latency; duplicates can accumulate because add_to_allowlist does not prevent them. While not exploitable by outsiders (admin-gated), it's an operational risk for mainnet scale.

Code (unchanged):

```
assert!(
    !lock.locked || vector::contains(&lock.allowlist, &sender_addr),
    EUnauthorized
);
```

```
let (found, index): (bool, u64) = vector::index_of(&lock.allowlist, &address);
assert!(found, EAddressNotFound);
vector::swap_remove(&mut lock.allowlist, index);
```

Recommendation:

- Switch to a Table<address, bool> for O(1) membership checks and reject duplicates in add_to_allowlist. If vector must remain, cap length, dedupe on insert, and emit allowlist size metrics in events to aid monitoring.

Implemented Fixes:

The CreatePoolLock struct still uses a vector for its allowlist, so the O(n) scan inefficiency remains. However, the add_to_allowlist function was improved to check for pre-existing addresses and prevent duplicates from being added , which resolves part of the original finding.

FINDINGS

Low Risk

L-01: Percentage rounding/dust in B splits

✅ RESOLVED

Description: The 30% reward calculation uses integer division; any remainder (“dust”) is implicitly pushed into the LP remainder, causing tiny drift from the advertised 40/30/30 split on certain totals.
Recommendation: Document the rounding policy and emit the computed amounts in the pool-created event. Optionally assert divisibility (e.g., by 10) or perform banker’s rounding to keep splits exact.
Analysis: This was fixed as part of the remediation for **H-04**. The new_lotus_pool function now includes assertions that enforce exact divisibility, ensuring that the 40/30/30 split results in no rounding dust being pushed to the LP portion .

L-02: No withdrawal events

✅ RESOLVED

Description: Administrative withdrawals of reserves (A, B, rewards, LP deposit) don’t emit events, reducing on-chain observability for indexers and alerts.
Recommendation: Emit explicit events per withdrawal with amounts and post-withdrawal balances.
Analysis: All administrative withdrawal functions (withdraw_balance_a, withdraw_balance_b, withdraw_rewards, withdraw_lp_deposit, withdraw_trade_fees) now emit corresponding events with the amount withdrawn and a timestamp, improving on-chain transparency.

L-03: Fee/role changes not evented

Description: Updates to global fee %, fee wallet, admin, and launch manager don’t emit events, making governance changes harder to track externally.
Recommendation: Emit ConfigUpdated-style events for each update path with old/new values.

L-04: Potential multiplication overflow guards missing (UX)

Description: Some parameter products (e.g., block_quantity * max_blocks, block_price * max_blocks) rely on Move’s default abort-on-overflow. While safe, failures appear as runtime aborts rather than user-friendly messages.
Recommendation: Pre-validate against sensible upper bounds and add specific error codes for clearer failures.

FINDINGS

Low Risk

L-05: “One per wallet” persists after sellback

✅ DESIGN CHOICE

Description: After selling back during buyback, the buyer remains flagged as having purchased and can’t re-purchase later. This may surprise users who interpret “sell back once” as “can re-enter later.”

Recommendation: Clarify UI copy and docs. If policy should allow re-entry, flip a per-wallet status on sellback (design choice).

Analysis: The contract logic that keeps a user in the buyers table after they sell back remains unchanged. As this is a user experience and policy decision rather than a security flaw, it is considered acknowledged.

L-06: Minor inconsistency/typos in comments and error docs

✅ RESOLVED

Description: Typos (e.g., “divisable”, “Despost”) and unused error notes can reduce auditability and confidence.

Recommendation: Clean up comments and remove dead references; keep error names consistent with checks to aid reviewers and tooling.

L-07: Trade fee rounding favors payer by truncation (policy clarity)

Description: Fee computation uses integer division (truncate). Users paying near thresholds may see a 1-unit difference vs. decimal expectations.

Recommendation: Document the “floor” rounding rule; include fee and total in the Trade event (already present) and mirror in frontend totals.

FINDINGS

Low Risk

L-08: Factory/registry observability gaps

Description: The factory cleanly enforces uniqueness of (A,B) pairs, but there are no events for “pool indexed/removed” beyond creation.

Recommendation: Emit a factory index event at creation (including typed pair) and, if you ever add deactivation, a corresponding removal event to help indexers maintain a canonical registry.

L-09: Small-sale usability constraints

Description: Very small total_b values can fail the “30% > 0” check, making micro-launches impossible.

Recommendation: Document a minimum viable supply or relax the check for micro sales while preserving invariant safety.

L-10: Anti-Sybil is outside smart-contract scope

Description: “One block per wallet” doesn’t prevent multi-address participation. That’s expected but worth calling out for launch fairness claims.

Recommendation: Communicate this clearly in docs; consider optional off-chain KYC/allowlist phases for launches that require stricter fairness.

FINDINGS

Informational

I-01: Public claims depend on off-chain SRM steps

Description: “LP permanently locked,” “100% of SUI to LP,” and reward burns/distributions are marketed outcomes but not enforced in this module. They require off-chain execution (migration to SRM, lock mechanics). This is informational for users/integrators to set expectations.

I-02: Rewardian randomness/mechanics not defined on-chain

Description: The “bonus rewards (1.5x–5x)” and whale/dolphin airdrop selection process aren’t implemented or specified in the contract. Any randomness or selection logic will be off-chain or in other contracts; auditors and users should not assume on-chain fairness guarantees from this module alone.

I-03: Token assumptions (no fee-on-transfer / standard decimals)

Description: Trade math presumes standard Sui `Coin<T>` without transfer fees, rebasing, or non-standard decimals. Using exotic token types for A/B may cause unexpected pricing or accounting drift. This is a design assumption rather than a vulnerability.

I-04: Single active pool per (A,B) pair by design

Description: The factory registers pools by `(payment A, launch B)` pair, effectively preventing multiple concurrent pools for the same pair. Relaunches must use a different pair or retire the prior pool. This impacts operational planning but is not a security risk.

I-05: No reentrancy surface under Move’s resource model

Description: The module performs no external callbacks during state mutation, and Move’s linear resource model prevents classic reentrancy. This is a positive note confirming a common class of attack is out of scope here.

I-06: Event schema lacks versioning/typed reasons

Description: Events are emitted for major transitions, but there’s no explicit versioning or typed “reason” fields. Indexers can still track states, but future schema changes may require off-chain adapters. This is informational for observability/tooling planning.

CONCLUSION

Status: Ready for Production

We are pleased to conclude this updated security assessment of the Lotus Launchpad. The development team has shown exceptional dedication to security and transparency, addressing every critical and high-risk issue with effective on-chain solutions. The initial centralization risks have been successfully eliminated, transforming the smart contract into a robust and trustless system that now programmatically enforces its core user protections.

The most severe vulnerabilities, such as unrestricted withdrawals and manual control over the launch lifecycle, have been fully resolved. The contract now guarantees that funds are secure during the sale, that the buyback window is governed by objective on-chain rules, and that the final 24-hour countdown is strictly enforced.

Further review of the off-chain architecture, including the AWS Lambda functions for reward distribution, confirmed a secure and well-designed backend process that complements the on-chain logic.

With these crucial fixes implemented and the intended design clarified, the Lotus V1 platform has met the high security standards required for a mainnet launch. We commend the Lotus team for their swift and thorough response to the audit findings and confirm that the project is secure and **ready for production**.

SPYWOLF

CRYPTO SECURITY

ABOUT US

We are a growing crypto security agency offering audits, KYCs and consulting services for some of the top names in the crypto industry.

- **OVER 700 SUCCESSFUL CLIENTS**
- **MORE THAN 1000 SCAMS EXPOSED**
- **MILLIONS SAVED IN POTENTIAL FRAUD**
- **PARTNERSHIPS WITH TOP LAUNCHPADS, INFLUENCERS AND CRYPTO PROJECTS**
- **CONSTANTLY BUILDING TOOLS TO HELP INVESTORS DO BETTER RESEARCH**

Audits | KYCs | dApps
Contract Development

To hire us, reach out to
contact@spywolf.co or
t.me/joe_SpyWolf

FIND US ONLINE

SPYWOLF.CO

@SPYWOLFNETWORK

@SPYWOLFNETWORK

Disclaimer

This report shows findings based on our limited project analysis, following good industry practice from the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, overall social media and website presence and team transparency details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report.
While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER:
By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice.
No one shall have any right to rely on the report or its contents, and SpyWolf and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (SpyWolf) owe no duty of care towards you or any other person, nor does SpyWolf make any warranty or representation to any person on the accuracy or completeness of the report.
The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and SpyWolf hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, SpyWolf hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against SpyWolf, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts, website, social media and team.
No applications were reviewed for security. No product code has been reviewed.