Jupiter Lend

Security Assessment

November 5th, 2025 — Prepared by OtterSec

Akash Gurugunti sud0u53r.ak@osec.io

Gabriel Ottoboni ottoboni@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Jupiter engaged OtterSec to assess the `flashloan`, `liquidity`, `vaults`, `oracle`, and `merkleDistributor` programs. This assessment was conducted between August 20th and November 1st, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 10 findings throughout this audit engagement.

In particular, we identified a vulnerability where the instruction responsible for the payback portion of a flashloan fails to validate that the loan was actually paid back, leading to a critical loss of funds (OS-JPL-ADV-00).

Additionally, we identified that the oracle program incorrectly interprets the zero fee case of SPL stake pools, leading to a division by zero (OS-JPL-ADV-01).

Furthermore, when calculating the exchange prices, an overflow may occur during the computation of the supply yield ratio due to large intermediate multiplications of high-scaled values in the liquidity program (OS-JPL-ADV-02).

We made recommendations for refactoring the codebase for better functionality, addressing inconsistencies, and ensuring adherence to coding best practices. (OS-JPL-SUG-03). We further advised including certain validation logic to mitigate potential security issues (OS-JPL-SUG-04).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Instadapp/fluid-contracts-solana. This audit was performed against f3d0b48, PR#79, and PR#86.

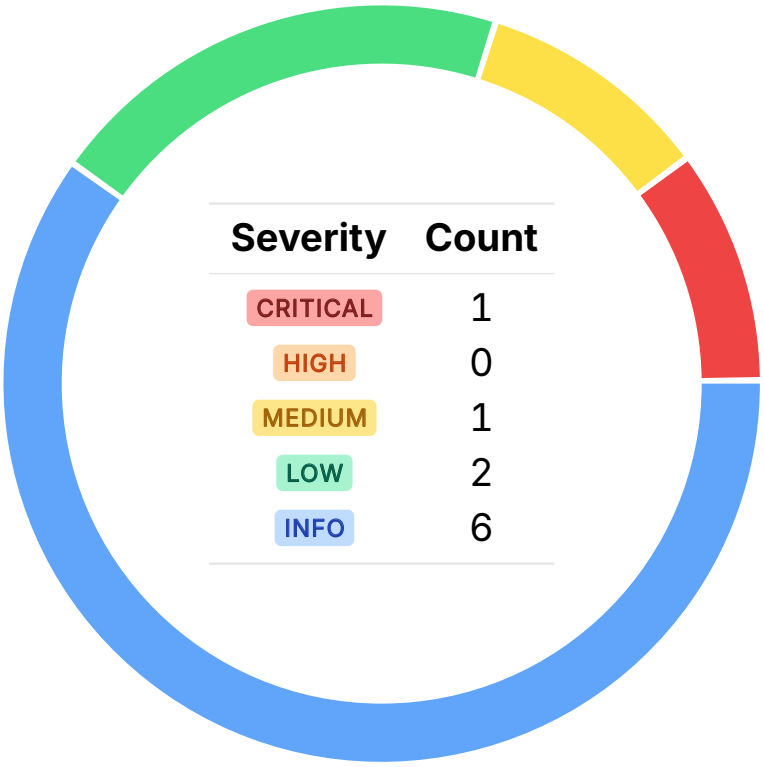
A brief description of the program is as follows:

Name	Description
flashloan	Enables borrowing and repaying tokens instantly within the same transaction through the protocol's liquidity layer, without requiring collateral.
liquidity	Manages on-chain lending and borrowing markets
vaults	Manages lending and borrowing operations through isolated vaults, maintaining global state, accounting, and exchange pricing.
merkleDistributor	Manages token reward distributions using Merkle tree proofs, allowing authorized roles to propose and approve reward roots, and enabling users to securely claim their rewards from a verified Merkle root.
oracle	Aggregates and normalizes price data from many on-chain sources.
merkle	Verifies Merkle proofs by recomputing the hash path from a leaf to the root using commutative Keccak256 hashing, ensuring a leaf's inclusion in a Merkle tree.

03 — Findings

Overall, we reported 10 findings.

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



Severity	Count
CRITICAL	1
HIGH	0
MEDIUM	1
LOW	2
INFO	6

04 — Vulnerabilities

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

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-JPL-ADV-00	CRITICAL	RESOLVED	`flashloan_payback` does not validate the `liquidity_program` account. This allows the load payback verification to by bypassed, leading to a critical loss of funds.
OS-JPL-ADV-01	MEDIUM	RESOLVED	`check_fee` in `stake_pool` incorrectly panics for valid zero-fee configurations `(0,0)` by dividing by zero, whereas Solana's stake-pool treats this as a legitimate zero-fee case.
OS-JPL-ADV-02	LOW	RESOLVED	`calculate_exchange_prices` may overflow during `ratio_supply_yield` computation due to large intermediate multiplications of high-scaled values.
OS-JPL-ADV-03	LOW	RESOLVED	The Pyth exponent is not properly validated, which may result in incorrect scaling or a panic from underflow.

Missing Account Validation CRITICAL OS-JPL-ADV-00

Description

In `flashloan_payback` within the `flashloan` program, there is no validation to make sure the `liquidity_program` account is indeed the liquidity program. This allows an attacker to set this account to a dummy program, such that, when `flashloan_payback` is called, the loan is never actually paid back. This breaks the instruction's core invariants, leading to a critical loss of funds.

>_ *flashloan/src/state/context.rs* RUST

```rust
#[derive(Accounts)]
pub struct Flashloan<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(mut)]
    pub flashloan_admin: Account<'info, FlashloanAdmin>,

    [...]

    /// CHECK: Safe, we check the address in the lending_admin PDA
    pub liquidity_program: UncheckedAccount<'info>,

    [...]
}
```

Remediation

Validate the `liquidity_program` account by verifying account `liquidity_program == flashloan_admin.liquidity_program`.

Patch

Resolved in e2e755f.

Handling of Zero Fee Case MEDIUM OS-JPL-ADV-01

Description

In `stake_pool::check_fee` within `oracle`, the logic does not handle the valid `numerator == 0 && denominator == 0` case that represents zero fees in Solana's native stake-pool implementation. In Solana's logic, such a configuration is explicitly supported, and the fee application function returns zero when the denominator is zero, and initialization ensures `numerator ≤ denominator`. However, `check_fee` performs a division by the denominator unconditionally, which panics if it is zero. This implies that legitimate zero-fee pools, such as Binance's, will incorrectly fail validation.

>_ *oracle/src/modules/stake_pool.rs* RUST

```rust
fn check_fee(fee: &Fee) -> Result<()> {
    let fee_scaled = fee.numerator.safe_mul(1000)?.safe_div(fee.denominator)?;

    if fee_scaled > MAX_FEE_CEILING {
        return err!(ErrorCodes::FeeTooHigh);
    }

    Ok(())
}
```

Remediation

Update the logic in `check_fee` to ensure `denominator == 0 && numerator == 0` represents the zero fees case.

Patch

Resolved in bc822ed.

Overflow in Supply Yield Ratio Calculation LOW OS-JPL-ADV-02

Description

`token_reserve::calculate_exchange_prices` in `liquidity` risks a `u128` overflow during the computation of `ratio_supply_yield`, due to the multiplication of large-scale values before any division is applied. Both `ratio_supply_yield` and `borrow_ratio` are scaled by 10^{17}, and when multiplied together with `FOUR_DECIMALS` (10^4), the intermediate result may reach close to the `u128` limit. This overflow may occur especially if there is more interest-free supply, resulting in `supply_ratio` becoming very small, but not zero, and inflating `ratio_supply_yield` values up to $\approx 10^{21}$.

>_ *programs/liquidity/src/state/token_reserve.rs* RUST

```rust
pub fn calculate_exchange_prices(&self) -> Result<(u128, u128)> {
    [...]
    ratio_supply_yield = ratio_supply_yield
        .safe_mul(borrow_ratio)?
        .safe_mul(FOUR_DECIMALS)?
        .safe_div(EXCHANGE_PRICE_RATE_OUTPUT_DECIMALS)?
        .safe_div(EXCHANGE_PRICE_RATE_OUTPUT_DECIMALS)?;
    [...]
    Ok((supply_exchange_price, borrow_exchange_price))
}
```

Remediation

Reorder operations during the computation of `ratio_supply_yield`, so that division by `EXCHANGE_PRICE_RATE_OUTPUT_DECIMALS` occurs earlier (after the first multiplication), to prevent high values due to intermediate multiplication.

Patch

Resolved in e4e06c4.

Unchecked Pyth Exponent Handling LOW OS-JPL-ADV-03

Description

`helper::read_rate_from_source` in `oracle` assumes that the Pyth price exponent is always non-positive and directly applies `abs`. Additionally, the code subtracts `abs(exponent)` from `RATE_OUTPUT_DECIMALS` without checking bounds. If the exponent is more negative than `-RATE_OUTPUT_DECIMALS`, this subtraction will panic due to underflow.

>_ programs/oracle/src/helper.rs RUST

```rust
fn read_rate_from_source<'info>([...]) -> Result<(u128, u128)> {
    let (rate, multiplier) = match source_info.source_type {
        SourceType::Pyth => {
            let pyth_price = read_pyth_source(&source, is_liquidate)?;
            let multiplier = u32::try_from(pyth_price.exponent.abs()).unwrap();
            let delta = 10u128.pow(RATE_OUTPUT_DECIMALS.safe_sub(multiplier.cast()?)?);

            (u128::try_from(pyth_price.price).unwrap(), delta)
        }
        _ => {
            return err!(ErrorCodes::InvalidSource);
        }
    };

    Ok((rate, multiplier))
}
```

Remediation

Validate that the Pyth exponent is `<= 0` before taking the absolute value, and check that it is `>= -RATE_OUTPUT_DECIMALS` before performing the subtraction.

Patch

Resolved in 509b76a.

05 — General Findings

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

ID	Description
OS-JPL-SUG-00	Token-2022 extensions may be modified after the creation of the `TokenReserve` account, enabling the extension authority to update the extension's parameters or reinitialize the mint with new extensions, bypassing the safety checks.
OS-JPL-SUG-01	`read_pyth_source` fails to validate that the `feed_id` in the incoming price update matches the expected `feed_id` for the account. If a pull feed were to be used, this could leave the program vulnerable to the feed unexpectedly changing.
OS-JPL-SUG-02	The oracle always reads `live[0]` instead of the latest transmission, which may result in the utilization of outdated prices when multiple live entries exist in the Chainlink feed.
OS-JPL-SUG-03	Recommendation for refactoring the codebase for better functionality, addressing inconsistencies and ensuring adherence to coding best practices.
OS-JPL-SUG-04	There are several instances where proper validation is not performed, resulting in potential security issues.
OS-JPL-SUG-05	The codebase contains multiple cases of redundant and unutilized code that should be removed for better maintainability and clarity.

Token-2022 Extension Validation Bypass OS-JPL-SUG-00

Description

The Token-2022 extension validation may be bypassed because it only checks the current configuration of extensions at initialization, not their mutability. Each extension has an associated authority that may update the extension's parameters after the creation of the `TokenReserve` account. Additionally, the `MintCloseAuthority` extension allows the mint to be closed and recreated with new, unsafe extensions.

Remediation

Ensure that for conditionally allowed extensions, the extension's authority is set to `None`. Furthermore, remove the `MintCloseAuthority` extension from the whitelist.

Alternatively, in addition to the above, implement a system such that whitelisted mints are allowed to use all extensions.

Missing Feed ID Validation

OS-JPL-SUG-01

Description

Currently, `read_pyth_source` reads price data from the given feed without confirming that the `feed_id` matches the intended feed. Since `feed_id` is not guaranteed to remain static in a pull feed, it is possible for an update to overwrite the account with data from a different feed, affecting downstream logic.

Since Jupiter Lend only uses push feeds, where the `feed_id` is guaranteed to not change, this has been acknowledged.

Remediation

Check that `price_message.feed_id` is the id of the expected feed.

Incorrect Latest Price Selection Logic OS-JPL-SUG-02

Description

Currently, the oracle's Chainlink reader always utilizes `live[0]` to fetch the price, assuming there is only one active transmission (`live_length = 1`). However, Chainlink feeds may maintain multiple `live` transmissions. Without utilizing `live_cursor` and `live_length` to find the latest entry, the program risks reading stale or outdated price data.

Remediation

Implement the same logic as done in `latest` in Chainlink store (shown below) to decide which index of `live` to access to retrieve the latest price. Alternatively, the SDK checks the live length.

>_ chainlink-solana/contracts/programs/store/src/state.rs RUST

```rust
pub fn latest(&self) -> Option<Transmission> {
    if self.header.latest_round_id == 0 {
        return None;
    }

    let len = self.header.live_length;
    // Handle wraparound
    let i = (self.header.live_cursor + len - 1) % len;

    Some(self.live[i as usize])
}
```

Patch

Resolved in bc822ed.

Code Refactoring

OS-JPL-SUG-03

Description

1. In the payback branch of `user::rebalance` in `vaults`, `borrow_to` should be set to `ctx.accounts.liquidity.key()` instead of the `rebalancer` address since the operation is a payback.

>_ programs/vaults/src/module/user.rs RUST

```
pub fn rebalance<'info>(
    ctx: Context<'_, '_, 'info, 'info, Rebalance<'info>>,
) -> Result<(i128, i128)> {
    [...]
    else if total_borrow_liquidity > total_borrow_vault {
        [...]
        accounts.operate_with_signer(
            OperateInstructionParams {
                [...]
                borrow_to: ctx.accounts.rebalancer.key(), // borrow_to will be rebalancer
                    ↪  address
                [...]
             },
             signer_seeds,
        )?;
    }
    [...]
}
```

2. The current implementation in `redstone::read_redstone_source` in `oracle` utilizes `write_timestamp` as the publish time, which may not accurately reflect when the price was actually computed. The `timestamp` field records the true calculation time and may be slightly earlier. For accuracy, the oracle should utilize the minimum of `timestamp` and `write_timestamp` as the effective `publish_time`.

>_ programs/oracle/src/modules/redstone.rs RUST

```
pub fn read_redstone_source(redstone_feed: &AccountInfo,is_liquidate: Option<bool>,) ->
    ↪  Result<Price> {
    [...]
    let price = get_price(&redstone_feed.value)?;
    let Some(publish_time) = redstone_feed.write_timestamp else {
        return err!(ErrorCodes::TimestampExpected);
    };
    [...]
}
```

3. `get_exchange_rate_for_hop` in `oracle::helper` utilizes `saturating_mul(multiplier)`, which silently caps the value at `u128::MAX` if an overflow occurs. This may result in distorted exchange rates due to the truncation. Utilize `checked_mul(multiplier)` instead to ensure the function fails on overflow.

>_ programs/oracle/src/helper.rs RUST

```rust
fn get_exchange_rate_for_hop<'info>([...]) -> Result<u128> {
    let (mut rate, multiplier) = read_rate_from_source(source, source_info,
        ↪  is_liquidate)?;

    rate = rate
        .saturating_mul(multiplier)
        .saturating_div(source_info.divisor);

    if rate > 0 && source_info.invert {
        rate = 10u128.pow(RATE_OUTPUT_DECIMALS * 2).saturating_div(rate);
    }

    rate = safe_multiply_divide(rate, current_hop_rate,
        ↪  10u128.pow(RATE_OUTPUT_DECIMALS))?;

    Ok(rate)
}
```

4. The current `#[max_len(10)]` annotation in the `state::MerkleAdmin` structure limits the `auths` vector to 10 authorized addresses. However, `MAX_AUTH_COUNT` is already defined to represent the maximum number of authorized accounts. Utilizing the hardcoded 10 instead of `MAX_AUTH_COUNT` creates a risk of inconsistency. Specifically, if `MAX_AUTH_COUNT` changes in the future, the structure's limit will not automatically update. Replace 10 with `MAX_AUTH_COUNT` for better maintainability and consistency.

>_ programs/merkleDistributor/src/state/state.rs RUST

```rust
#[account]
#[derive(InitSpace)]
pub struct MerkleAdmin {
    pub authority: Pubkey,

    #[max_len(10)]
    pub auths: Vec<Pubkey>,
    [...]
}
```

Remediation

Incorporate the above refactors.

Patch

Resolved in e4e06c4, bc822ed, and 7303de5.

Missing Validation Logic

OS-JPL-SUG-04

Description

1. In `admin::init_distributor` within `merkleDistributor`, if `cycle_in_hours > distribution_in_hours`, the division truncates to zero, rendering `cycles_per_distribution == 0`. This creates an invalid distributor setup where no reward cycles exist, breaking the reward allocation. Validate that `cycle_in_hours` is less than `distribution_in_hours` before computing.

>_ programs/merkleDistributor/src/module/admin.rs RUST

```rust
pub fn init_distributor(ctx: Context<InitDistributor>, params: InitializeParams) ->
    ↪  Result<()> {
    [...]
    let cycles_per_distribution = params
        .distribution_in_hours
        .cast::<u128>()?
        .safe_div(params.cycle_in_hours.cast()?)?
        .cast()?;
    [...]
}
```

2. `oracle::init_admin` incorrectly checks `oracle_admin.authority != Pubkey::default` instead of validating the `authority` parameter. Since `oracle_admin` is always freshly initialized, this check will always pass, even if the provided authority is invalid. This implies a zero `pubkey` (`Pubkey::default`) may be set as the admin, leaving the oracle without a valid authority. Explicitly check `authority == Pubkey::default` and throw an error in that case.

>_ programs/oracle/src/lib.rs RUST

```rust
pub fn init_admin(ctx: Context<InitAdmin>, authority: Pubkey) -> Result<()> {
    let oracle_admin = &mut ctx.accounts.oracle_admin;

    if oracle_admin.authority != Pubkey::default() {
        return err!(ErrorCodes::InvalidParams);
    }

    oracle_admin.authority = authority;
    oracle_admin.auths.push(authority);

    Ok(())
}
```

Remediation

Include the listed validations.

Patch

Resolved in 7303de5 and 0cc0202.

Unutilized/Redundant Code

OS-JPL-SUG-05

Description

1. `admin::update_role` in `merkleDistributor` only blocks redundant deactivation (turning off an inactive role) but still allows redundant activation (turning on an already active role). This results in unnecessary state changes and misleading event emission. Prevent activating an already active role.

>_ *programs/merkleDistributor/src/module/admin.rs* RUST

```rust
pub fn update_role(ctx: Context<UpdateRole>, turn_off: bool) -> Result<()> {
    let role_account = &mut ctx.accounts.role_account;

    if turn_off && !role_account.active {
        return Err(ErrorCodes::InvalidParams.into());
    }

    role_account.active = !turn_off;
    emit!(LogUpdateRole {
        address: role_account.address,
        role: role_account.get_role(),
    });

    Ok(())
}
```

2. `merkle_admin` account in the `ProposeRoot` and `ApproveRoot` structure is never utilized in the instruction logic, rendering it redundant.
3. In `user::liquidate` in `vaults`, there is a double borrow panic because `vault_state` is already mutably borrowed once via `load_mut`, and calling `ctx.accounts.vault_state.load` again tries to immutably borrow it. Utilize `vault_state.topmost_tick` directly instead of reloading it.
4. The `signer` account in `context::UpdateExchangePrices` structure within `vaults` is not utilized anywhere and may be removed.

>_ *programs/vaults/src/state/context.rs* RUST

```rust
#[derive(Accounts)]
pub struct UpdateExchangePrices<'info> {
    pub signer: Signer<'info>,
    [...]
}
```

Remediation

Remove the redundant and unutilized code instances highlighted above.

Patch

Resolved in e4e06c4.

A — Vulnerability Rating Scale

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

CRITICAL

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

Examples:

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

HIGH

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

Examples:

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

MEDIUM

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

Examples:

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

LOW

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

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

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

Examples:

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

B — Procedure

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

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

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

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

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

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

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