Bluefin Spot

Security Assessment

November 2nd, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Michał Bochnak embe221ed@osec.io

Sangsoo Kang sangsoo@osec.io

Table of Contents

01 — Executive Summary

Overview

Firefly protocol engaged OtterSec to assess the `bluefin-spot` program. This assessment was conducted between October 22nd and November 1st, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 9 findings throughout this audit engagement.

In particular, we identified a critical vulnerability where the constant for storing the maximum value of a 64-bit unsigned number is incorrectly defined, as it is missing a character in its declaration, resulting in a length of 15 instead of the expected 16 characters, which may have adverse effects on the tick calculation (OS-RPL-ADV-00). Additionally, when a reward distribution is restarted after a pause, the system incorrectly includes the inactive periods in reward calculations, resulting in inaccurate reward distribution (OS-RPL-ADV-01). Furthermore, the configuration module lacks a function to update the version, which is essential for managing package upgrades and ensuring compatibility with newer versions (OS-RPL-ADV-02).

We also recommended including validation during the pool creation process to ensure that the initial square root price falls within a safe and operational range (OS-RPL-SUG-00), and advised incorporating additional checks within the codebase for improved robustness and security (OS-RPL-SUG-02). We further suggested modifying the codebase for improved functionality, efficiency, and maintainability (OS-RPL-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/fireflyprotocol/bluefin-spot-contracts. This audit was performed against 68beb25.

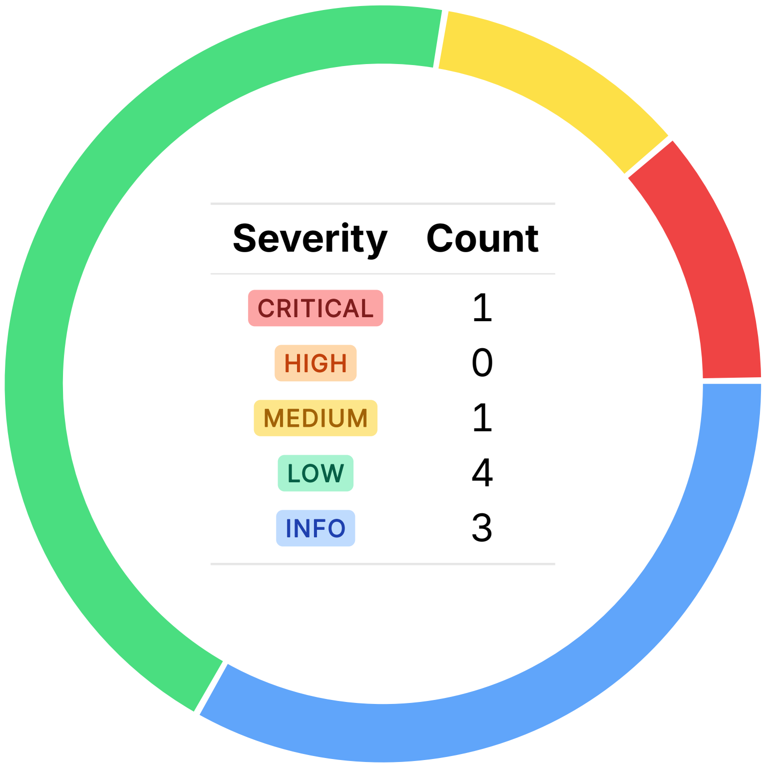
A brief description of the program is as follows:

Name	Description
bluefin-spot	This module includes the Sui smart contracts for the Bluefin spot exchange program.

03 — Findings

Overall, we reported 9 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	4
INFO	3

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-RPL-ADV-00	CRITICAL	RESOLVED ✓	The `MAX_u64` constant in the constants module is missing an `f`, resulting in a length of 15 instead of the expected 16 characters.
OS-RPL-ADV-01	MEDIUM	RESOLVED ✓	When a reward distribution is restarted after a pause, the system incorrectly includes the inactive periods in the reward calculations, resulting in inaccurate reward distribution.
OS-RPL-ADV-02	LOW	RESOLVED ✓	The configuration module lacks a function to update the version, which is essential for managing package upgrades and ensuring compatibility with newer versions.
OS-RPL-ADV-03	LOW	RESOLVED ✓	The flash swap operation is vulnerable to reentrancy attacks.
OS-RPL-ADV-04	LOW	RESOLVED ✓	In `update_pool_state`, the `oracle::update` call utilizes the updated `current_tick_index` after the swap, which may result in inaccurate oracle observations.
OS-RPL-ADV-05	LOW	RESOLVED ✓	`swap_in_pool` contains a vulnerability where the conditions for validating `sqrt_price_max_limit` utilize `<=` and `>=`, allowing for invalid swaps that exceed the tick boundaries.

Faulty Constant Definition CRITICAL OS-RPL-ADV-00

Description

The constant `MAX_u64` is supposed to represent the maximum value of a 64-bit unsigned integer, which is $2^{64} - 1$. In hexadecimal, this value should be represented as `0xFFFFFFFFFFFFFFFF`, which consists of 16 hexadecimal characters. However, in the current declaration, there is an `f` missing, resulting in the length of the constant being 15 characters instead of the required 16.

>_ *sources/maths/bit_math.move* MOVE

```
public fun least_significant_bit(mask: u256) : u8 {
    assert!(mask > 0, 0);
    let bit = 255;
    [...]
    if (mask & (constants::max_u64() as u256) > 0) {
        bit = bit - 64;
    } else {
        mask = mask >> 64;
    };
    [...]
    bit
}
```

Since `MAX_u64` is incorrectly defined, it results in improper bitwise operations when determining the least significant bit (LSB) of the input mask in `least_significant_bit`. Specifically, the function checks if `mask & max_u64` is greater than 0 to determine if any bits in the least significant 64 bits are set. If this constant is one character short, it effectively ignores the highest bit (the most significant bit in the 64-bit range) when performing this check. This is particularly critical because this function is utilized to search for the next initialized tick, and thus, it may result in the calculation of an incorrect tick position.

Remediation

Correct the definition of `MAX_u64` constant to include the missing character `f`, ensuring it has the correct value and length of 16 characters.

Patch

Resolved in f9025e9.

Reward Accumulation During Inactive Time Period MEDIUM OS-RPL-ADV-01

Description

There is a vulnerability in how reward calculations are handled for liquidity positions after a reward distribution has ended and then restarts. There is no way to correctly restart the distribution of the same type after a reward distribution has finished. Specifically, the calculation incorrectly includes inactive time (the time period after the distribution ended but before it restarts) in the reward accumulation, resulting in inaccurate rewards for positions.

>_ *sources/pool.move* MOVE

```move
public(friend) fun update_reward_infos<CoinTypeA, CoinTypeB>(pool: &mut Pool<CoinTypeA,
    ↪  CoinTypeB>, current_timestamp_seconds: u64) : vector<u128> {
    let reward_growth_globals = vector::empty<u128>();
    let current_index = 0;
    while (current_index < vector::length<PoolRewardInfo>(&pool.reward_infos)) {
        [...]
        if (current_timestamp_seconds > reward_info.last_update_time) {
            [...]
            if (pool.liquidity != 0 && min_timestamp > reward_info.last_update_time) {
                    let rewards_accumulated = full_math_u128::full_mul(((min_timestamp
                        ↪  -reward_info.last_update_time) as u128),
                        ↪  reward_info.reward_per_seconds);
                    [...]
                    reward_info.total_reward_allocated = reward_info.total_reward_allocated +
                        ↪  ((rewards_accumulated/ (constants::q64() as u256)) as u64);
            };
            reward_info.last_update_time = current_timestamp_seconds;
        };
        vector::push_back<u128>(&mut reward_growth_globals, reward_info.reward_growth_global);
    };
    reward_growth_globals
}
```

In both `pool::update_reward_infos` and `position::update`, reward growth is based on the difference between the `current_timestamp_seconds` and the last update time. When `pool::update_pool_reward_emission` is called to restart distribution, the function does not account for any time gap between the previous reward end and the new start time. As a result, liquidity providers may receive extra rewards that do not correspond to any actual activity.

For example, if reward distribution ends at time $t0$, and a new distribution restarts at time $t1$, the reward accumulation for liquidity positions incorrectly includes the inactive period $[t0, t1]$. Ideally, rewards should only accumulate while there is an active reward emission. However, because `update_reward_infos` and `update` rely on the difference between the last updated timestamp and the current timestamp, they end up counting this inactive interval in reward calculations.

Remediation

When calculating rewards, ensure that only the time intervals where rewards were actively distributed are included.

Patch

Resolved in f9025e9.

Absence of Version Update Functionality LOW OS-RPL-ADV-02

Description

In `config`, there is no mechanism to update the protocol's version during package upgrades. This limitation will pose significant issues when deploying new versions of the protocol, especially if breaking changes are introduced. The `GlobalConfig` structure contains a `version` field, which is intended to track the current version of the protocol. The `VERSION` constant is defined in the module, but there is no function to modify the `version` field of `GlobalConfig` after the initial setup.

Remediation

Implement a functionality to upgrade the `version` field within `config`.

Patch

Resolved in f9025e9.

Risk of Reentrancy During Flash Swap LOW OS-RPL-ADV-03

Description

In `pool`, there is a lack of a `reentrancy` guard during flash swap operations, allowing potential reentrant calls to be made via other functions, which might result in the manipulation of the pool values.

Remediation

Add a reentrancy guard to prevent the calling of other functions during the execution of a flash swap.

Patch

Resolved in f9025e9.

Improper Oracle Update LOW OS-RPL-ADV-04

Description

`pool::update_pool_state` is responsible for updating the state of the liquidity pool after a swap operation occurs. In the current code, `oracle::update` is called after the pool's `current_tick_index` and `current_sqrt_price` have been updated based on the swap result. This timing will result in the recording of inaccurate and misleading data in the oracle regarding the pool's state before the swap occurred, rendering the data inconsistent.

>_ *sources/pool.move* MOVE

```move
fun update_pool_state<CoinTypeA, CoinTypeB>(pool: &mut Pool<CoinTypeA, CoinTypeB>, swap_result:
    ↪  SwapResult, current_time: u64) {
    // current tick index of pool is not the same as swap result
    if (!i32::eq(pool.current_tick_index, swap_result.current_tick_index)) {
        pool.current_sqrt_price = swap_result.end_sqrt_price;
        pool.current_tick_index = swap_result.current_tick_index;
        oracle::update(
            &mut pool.observations_manager,
            pool.current_tick_index,
            pool.liquidity,
            current_time,
        );
    } else {
        pool.current_sqrt_price = swap_result.end_sqrt_price;
    };
    [...]
}
```

Remediation

Call `oracle::update` before updating `current_tick_index`.

Patch

Resolved in f9025e9.

Incorrect Price Boundary Checks LOW OS-RPL-ADV-05

Description

In `pool::swap_in_pool`, for a swap from `CoinTypeA` to `CoinTypeB` (`a2b` is true), the current square root price of the pool must be greater than `sqrt_price_max_limit`, and `sqrt_price_max_limit` must be greater than or equal to the minimum square root price. For a swap from `CoinTypeB` to `CoinTypeA` (`a2b` is false), the current square root price must be less than `sqrt_price_max_limit`, and `sqrt_price_max_limit` must be less than or equal to the maximum square root price.

>_ *sources/pool.move* MOVE

```move
fun swap_in_pool<CoinTypeA, CoinTypeB>(
    clock: &Clock,
    pool: &mut Pool<CoinTypeA, CoinTypeB>,
    a2b: bool,
    by_amount_in: bool,
    amount:u64,
    sqrt_price_max_limit: u128): SwapResult
{
    [...]
    if (a2b) {
        assert!(pool.current_sqrt_price > sqrt_price_max_limit && sqrt_price_max_limit >=
            ↪  tick_math::min_sqrt_price(), errors::invalid_price_limit());
    } else {
        assert!(pool.current_sqrt_price < sqrt_price_max_limit && sqrt_price_max_limit <=
            ↪  tick_math::max_sqrt_price(), errors::invalid_price_limit());
    };
    [...]
}
```

However, if the current tick is at `MIN_TICK` (the lowest price in the pool), then the condition `sqrt_price_max_limit >= min_sqrt_price` will still pass if `sqrt_price_max_limit` equals `min_sqrt_price`. This implies that the swap may attempt to process even when the price boundary is effectively breached. Similarly, if the current tick is at `MAX_TICK - 1`, utilizing `<=` may allow `sqrt_price_max_limit` to equal `max_sqrt_price`, which will result in an invalid operation that tries to push the square root price beyond the maximum limit.

Thus, utilizing `>=` and `<=` in the boundary checks is not appropriate, as allowing the price to hit the exact minimum or maximum boundaries will result in attempts to execute swaps that lead to invalid price states.

Remediation

Modify the conditions to utilize `<` and `>` instead of `<=` and `>=`. This ensures that the square root price limit is strictly greater than the minimum square root price and strictly less than the maximum square root price.

Patch

Resolved in f9025e9.

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-RPL-SUG-00	The pool may be initialized at `max_sqrt_price`, but `max_tick` is an area that should remain unreachable.
OS-RPL-SUG-01	Recommendation for modifying the codebase for improved functionality, efficiency, and maintainability.
OS-RPL-SUG-02	There are several instances where proper validation is not done, resulting in potential security issues.

Initialization Price Validation OS-RPL-SUG-00

Description

In `tick_math::get_tick_at_sqrt_price`, utilizing `<=` implies that the pool may be initialized at `max_sqrt_price`. However, `max_tick` represents an area that should remain unreachable. Thus, allowing initialization at `max_sqrt_price` effectively opens up the possibility for the pool to operate at extreme prices.

>_ *sources/maths/tick_math.move* MOVE

```
public fun get_tick_at_sqrt_price(sqrt_price: u128): i32::I32 {
    assert!(sqrt_price >= MIN_SQRT_PRICE_X64 && sqrt_price <= MAX_SQRT_PRICE_X64,
        ↪  EINVALID_SQRT_PRICE);
    let r = sqrt_price;
    [...]
}
```

Remediation

Include validation during the pool creation process to ensure that the initial `sqrt_price` falls within a safe and operational range.

Patch

Resolved in f9025e9.

Code Refactoring OS-RPL-SUG-01

Description

1. For improved functionality and control, add functions to remove an address from the `reward_managers` list in `config`, and to update the `is_paused` state of the pool.
2. Include the `entry` keyword in `gateway::collect_reward` to allow the function to be called externally, enabling users to easily retrieve their rewards from their liquidity positions in the pool.

>_ *sources/gateway.move* MOVE

```move
/// Allows user to collect the rewards accrued on their position
public fun collect_reward<CoinTypeA, CoinTypeB, RewardCoinType>(
    clock: &Clock,
    protocol_config: &GlobalConfig,
    pool: &mut Pool<CoinTypeA, CoinTypeB>,
    position: &mut Position,
    ctx: &mut TxContext
    ){
    [...]
}
```

Remediation

Implement the above-mentioned suggestions.

Patch

Resolved in f9025e9.

Missing Validation Logic OS-RPL-SUG-02

Description

1. in `add_liquidity`, `add_liquidity_with_fixed_amount`, `remove_liquidity`, `swap`, and `flash_swap`, check that the amount passed is greater than zero, as allowing zero amounts to be passed into these functions will result in nonsensical operations.
2. Limit the values of `fee_rate` and `tick_spacing` to an acceptable range in `pool::new`. As a high fee rate will discourage users from providing liquidity or trading within the pool, and if `tick_spacing` is set too small, it may result in inefficient liquidity distribution, rendering it difficult for traders to execute orders at desired prices.
3. In `admin::update_supported_version`, add a validation to ensure the incremented `config.version` does not exceed the latest version.

Remediation

Incorporate the above validations.

Patch

Resolved in f9025e9.

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.