🌱Hyacinth Move Audit Report🌱

1. Improper owner access control

Severity: Critical
Likelihood: High
Location: Ln - public fun cancel_order

Description:

The cancel_order function does not make any sort of assertion that signifies that the signer is the owner of the order before being able to cancel the order and transfer assets to the caller.

Correct Implementation:

```
assert!(order.user_address == address_of(user), ERR_PERMISSION_DENIED);
```

This ensures that the caller is the owner of the order.

2. Absence of Generics checking

Severity: Critical
Likelihood: High
Location: Ln - public fun cancel_order

Description:

The cancel_order function does not assert that the generic type BaeCoinType matches the base_type stored in the Order resource. This function enables unlock of liquidity for a base coin type, so a malicious actor could take advantage of this lack of check present to drain all the liquidity from the AMM. A malicious actor can place a limit swap order and cancel the order passing the incorrect coin type.

Correct Implementation:

```
assert!(order.base_type == type_info::type_of<BaseCoinType>(), ERR_ORDER_WRONG_COIN_TYPE);
```

3. Arithmetic Precision Errors

Severity: Medium
Likelihood: High
Location: *Ln* - public fun calculate_protocol_fees

Description:

In the calculate_protocol_fees function, the fees are calculated by taking the percentage of the order size. The issue is, if the size is less than the given 10000/PROTOCOL_FEE_BPS, then this will round down to 0. So in essence, a user can bypass fees when performing operations in this protocol such as swapping and removing liquidity.

Correct Implementation:

Have an implementation that requires the size of the order to be greater than the minimum amount and set the protocol fees to be calculated to an amount greater than zero.

4. Unbounded Execution - DOS

Severity: Critical
Likelihood: High
Location: *Ln -In the get_order_by_id , drop_order, and fullfill_order functions are susceptible to DOS. In the get_order_by_id, cancel_order, and fulfill_order can be called to perform this vulnerability. In the fulfill_orders function, can be called by add_liquidity. In the drop_order function, which can be called by cancel_order and execute_limit_order*

Description:

These functions can lead to unbounded execution because they iterate over potentially large lists. An attacker could exploit this by registering a large number of orders, causing the functions to block and leading to denial-of-service (DOS).

1. An attacker could permanently block all users from canceling or fulfilling limit orders, locking funds in the protocol permanently.
2. An attacker could permanently block users from swapping, adding liquidity, and creating limit swap orders.

Correct Implementation:

1. Avoid looping over every order.
2. Limit the number of iterations each loop can perform.
3. Structure fees to incentivize users to fulfill each other's orders.

5. Manipulable Price Oracle resulting in a drain of the pool

Severity: Critical
Likelihood: High
Location: Ln -In the fun get_price_internal. This is where the price calculator mechanism lies.

Description:

The contract uses the ratio of the liquidity sizes of the tokens to determine the value of the liquidity token. The contract uses the ratio of the liquidity sizes of the tokens in the pool to calculate their prices, which an attacker can manipulate to drain the pool.

Correct Implementation:

To mitigate this vulnerability, use an external price oracle that cannot be easily manipulated. This oracle should provide reliable and tamper-resistant price data. Additionally, the internal price calculation should be augmented with checks against the external oracle to prevent such manipulations.

6. No check for account registration for Coin

Severity: Medium
Likelihood: High

Location: Ln *public fun limit_swap<BaseCoinType, QuoteCoinType>*

Description:

The execute_limit_order function fails to verify if the recipient account is registered to receive the quote coin, leading to potential issues during order execution. This function, called by execute_order and subsequently by fulfill_orders, is also indirectly invoked by add_liquidity. If an attacker creates an order targeting an unregistered account for the quote coin, it can block the execution of swaps and the addition of liquidity, thereby disrupting the contract's normal operation.

Correct Implementation:

Add the following to the limit_swap function
```
coin::register<BaseCoinType>(user);
coin::register<QuoteCoinType>(user);
```

7. Arithmetic Errors –Overflow

Severity: Medium
Likelihood: Medium
Location: Ln -
fun calculate_lp_coin_amount_internal:

- size * get_usd_value_internal(order_store, type)

fun calculate_protocol_fees:

- size * PROTOCOL_FEE_BPS / 10000

Description:

The calculate_lp_coin_amount_internal and calculate_protocol_fees functions in the contract are susceptible to overflow errors when handling large input sizes. This vulnerability can be exploited by creating orders with sizes large enough to cause arithmetic overflow, leading to a denial of service as operations like add_liquidity and fulfill_orders would fail.

Correct Implementation:

Cast operands to u128 before multiplication and ensure the result fits within u64 limits to prevent overflow errors.

8. Misuse of Resource management

Severity: Medium
Likelihood: Low
Location: Ln - struct OrderStore has key {}

- *orders: vector<Order>,*

Description:

In this module, the Order resources in the orders vector are stored on the @buggyswap. This is considered a bad practice. If the vector of orders grows too large then a further iteration can cause out-of-gas abort acting as a denial of service

Correct Implementation:

Store resources within the user's account.

9. Business logic flaw

Severity: High
Likelihood: Medium
Location: Entire protocol

Description:

In this project, users have no incentive to provide liquidity to the AMM. Usually, there is some sort of incentive in place.

Correct Implementation:

AMM protocols normally have some type of incentive. These can come in the form of fee collection for swap or liquidity-changing operations. Ideally, this contract should implement some sort of fee structure that would incentivize different operations.

10. Fulfill_orders Borrows post Extraction

Severity: Medium
Likelihood: High
Location: Entire protocol

Description:

The Order is extracted from order_option and then borrowed again to fetch the order ID, which causes an error because the Order is no longer present in the Option after extraction. If any limit order is successfully fulfilled during a fulfill_orders call, the transaction will abort, potentially preventing users from being able to add liquidity.

Correct Implementation:

Extract the order from the Option once or borrow it twice correctly before extraction.

11. Incorrect Visibility in remove_liquidity Function

Severity: High
Likelihood: High
Location: **remove_liquidity function**

Description:

The remove_liquidity function in this module is intended to allow users to withdraw their liquidity from the pool. However, this function is currently private, meaning it cannot be called by external users. As a result, users are unable to withdraw their liquidity, which is not the intended behavior. The function remove_liquidity has default visibility (private), preventing users from calling it. This restriction hinders users from accessing their funds, leading to a poor user experience and potentially causing trust issues with the platform.

Correct Implementation:

To address this vulnerability, the remove_liquidity function should be declared as public. This change will allow users to call the function and withdraw their liquidity as intended.

12. Improper admin verification

Severity: Critical
Likelihood: High
Location: **fun admin_create_coinstore**

Description:

The admin_create_coinstore function in this module is designed to allow only the admin to create a coinstore for a specific coin type. However, if the check to verify the admin's address is removed or bypassed, anyone could potentially call this function, leading to the unauthorized creation of coinstores and potential misuse of the system.

Correct Implementation:

The following line ensures that only the admin (identified by the address @buggyswap) can call the admin_create_coinstore function:

```
assert!(address_of(admin) == @buggyswap, ERR_PERMISSION_DENIED);
```

If this check is removed, anyone could call admin_create_coinstore and create coinstores for any coin type without proper authorization.

13. Unauthorized Protocol Control

Severity: Critical
Likelihood: High
Location: pause_protocol & resume_protocol functions

Description:

The pause_protocol and resume_protocol functions in the Move module are vulnerable to unauthorized access, allowing any user to pause or resume the protocol. This vulnerability arises due to the lack of access control checks within these functions.

Denial of Service (DoS): Any user can pause the protocol, potentially causing a denial of service by preventing legitimate operations from being processed.

Operational Disruption: Unauthorized users can resume the protocol, potentially disrupting maintenance operations or other planned pauses.

Correct Implementation:

```
public entry fun pause_protocol(admin: &signer) {
    // Ensure only the authorized admin can pause the protocol
    assert!(address_of(admin) == @admin_address, ERR_PERMISSION_DENIED);
    let state = borrow_global_mut<State>(@buggyswap);
    state.is_paused = true;
}

public entry fun resume_protocol(admin: &signer) {
    // Ensure only the authorized admin can resume the protocol
    assert!(address_of(admin) == @admin_address, ERR_PERMISSION_DENIED);
    let state = borrow_global_mut<State>(@buggyswap);
    state.is_paused = false;
```

}

14. Borrow After Extract Vulnerability

Severity: Medium
Likelihood: Medium
Location: **fulfill_orders function.**

vector::push_back(&mut successful_order_ids, option::borrow(&mut order_option).id);

This line attempts to borrow from order_option after it has been extracted earlier in the code:

let status = execute_order<CoinType>(order_store, &option::extract(&mut order_option));

After the option::extract(&mut order_option) call, order_option is empty, and attempting to borrow from it results in an error.

Description

In the fulfill_orders function, the code tries to use (borrow) an order from an Option after it has already been extracted from the Option. This results in an error because once you extract a value from an Option, the Option no longer contains that value, and trying to access it again will fail.

Correct Implementation:

Extract the Order Once: Extract the order from the Option a single time and store it in a variable for use, avoiding multiple borrow attempts after extraction.

Test Coverage: Ensure that this code is thoroughly tested, particularly for cases where status == 0. Proper test coverage would have detected this issue early on.