IRONNODE

ironnode.io

Kira Liquidity Pool Locker

Smart Contract Security Audit

by **IronNode** from **September 2 - September 24, 2024**

Executive overview

1 Executive overview

1.1 Introduction

The LP Locker is a smart contract platform designed to enhance trust and security in decentralized finance (DeFi) projects by allowing project owners to lock their Liquidity Pool (LP) tokens. This mechanism serves as a crucial step for projects aiming to establish themselves as safe and trustworthy within the DeFi ecosystem.

Key features of the LP Locker include:

- Three different timing options for locking LP tokens
- An option to extend the lock duration
- Transparency for token holders to view remaining lock duration
- A 1% tax on LP token locking transactions

By locking their LP tokens, project owners can demonstrably commit to not removing liquidity abruptly (often referred to as "rugging"), thereby fostering trust between the project teams and their token holders. This transparency allows holders to monitor lock durations and encourage project owners to extend locks when necessary.

In response to a request from the KIRA team, our security team conducted a thorough audit of the LP Locker contract to ensure its integrity, security, and reliability for both project owners and token holders in the broader DeFi community.

1.2 Audit summary

Our security audit team was allocated time to conduct an extensive review of the LP Locker contract. The audit was led by experienced security engineers with expertise in blockchain technology, smart contract security, and Rust programming.

1.2.1 Objectives of the audit

- Lock Mechanism Integrity: Evaluate the time-locking feature to confirm it operates as intended and offers the three timing options securely.
- Extension Functionality: Verify the lock extension feature works correctly and can only be triggered by authorized parties.
- Transparency Features: Assess the mechanisms that allow token holders to view remaining lock durations.
- Tax Implementation: Confirm the 1% tax on LP token locking is implemented correctly and securely.
- Token Handling Security: Assess the contract's mechanisms for handling both native and CW20 tokens to ensure they are secure and free from vulnerabilities.
- Access Control: Review the access control mechanisms to ensure only authorized parties can perform sensitive operations.
- Mathematical Operations: Examine all mathematical operations for potential overflow, underflow, or other arithmetic vulnerabilities.
- Data Storage Efficiency: Assess the efficiency and security of data storage methods used in the contract.

1.3 Scope

1.3.1 Code repositories

Kira_SC_LP_Locker

Branch: main

Commit Hash: a8b560d4fc37d9226a3fb89e20ab9dfd1915f636

Remediation Commit ID

8f294c482cc43d987d7e3d9777f20805fd289622

1.4 Assessment summary & findings overview

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	1	3	0

Security analysis	Risk Level	Remediation
Lack of Access Control in `execute_unstake`	HIGH	SOLVED
Unchecked Mathematical Operations	HIGH	SOLVED
Unused `native_token` Parameter	MEDIUM	SOLVED
Inconsistent Token Handling in `get_token_account` and `transfer_token_message`	LOW	SOLVED
Use of Inline Static Values	LOW	SOLVED
Inefficient ID Format and Storage	LOW	SOLVED

Application Flow Analysis

2 Kira Liquidity Pool Locker

This section of the report provides an in-depth analysis of the application flow for the Kira Liquidity Pool Locker contract. The analysis covers the logical flow of the code, with an emphasis on two primary roles: User and Admin.

2.1 User Role

The user role encompasses wallet addresses that interact with the LP Locker contract. Users can lock their LP tokens, extend the lock time, and unstake their tokens after the lock period expires.

2.1.1 Functions

- execute_receive_liquidity: Allows users to lock their LP tokens. This function is called when the contract receives CW20 tokens. It creates a new LiquidityPool entry with the specified lock time. A 1% fee is deducted from the locked amount.
- execute_extend_locktime: Allows users to extend the lock time of their existing LiquidityPool. Users can only extend their own pools.
- execute_unstake: Allows users to withdraw their locked tokens after the lock period has expired. This function can currently be called by any user, which is a security concern that needs to be addressed.

2.2 Admin Role

The admin role is responsible for managing the configuration settings of the LP Locker contract. The admin can update various parameters of the contract.

2.2.1 Functions

- execute_update_config: Allows the admin to update the contract's configuration. This includes: Setting the native token (currently unused), updating the fee address, changing the fees percentage, enabling or disabling the contract.

2.3 Query Functions

The contract also provides several query functions that allow users and external systems to retrieve information:

- query_config: Returns the current configuration of the contract, including the owner address, whether the contract is enabled, and the fees percentage.
- query_liquidities: Retrieves a list of all LiquidityPools, or optionally filtered by a specific owner address.
- query_liquidity: Retrieves details of a specific LiquidityPool by its ID.

Findings

4 Findings

4.1 Lack of Access Control in `execute_unstake`

The `execute_unstake` function can be called by any user, potentially allowing unauthorized unstaking, if the time condition met.

4.1.1 Code location

```
src/contract.rs
```

4.1.2 Recommendation

Implement access control for the `execute_unstake` function. Only allow the stake owner or authorized addresses to call this function.

4.2 Unchecked Mathematical Operations

The contract uses unchecked mathematical operations on multiple locations, which could lead to panics due to overflow or underflow.

4.2.1 Recommendation

Use checked mathematical operations to handle potential overflow/underflow gracefully.

An example code location:

```
let fee_amount = (amount * Uint128::from(fee)) / Uint128::from(100u64);
let new_amount = amount - fee_amount;
```

Can be handled like this:

```
let fee_amount = amount
    .checked_mul(Uint128::from(fee))
    .map_err(|_| ContractError::ArithmeticErr {})?
    .checked_div(Uint128::from(100u64))
    .map_err(|_| ContractError::ArithmeticErr {})?;
let new_amount = amount
    .checked_sub(fee_amount)
    .map_err(|_| ContractError::ArithmeticErr {})?;
```

4.3 Unused `native_token` Parameter

The `native_token` parameter is included in the instruction data but is not being saved, edited, or utilized in any meaningful way. It's also not forwarded to the response.

4.3.1 Code location

src/util.rs (Lines 28-55)

```
pub fn execute_update_config(
    storage: &mut dyn Storage,
    address: Addr,
    native_token: String,
    fee_address: Addr,
    fees_percentage: u64,
    is_enabled: bool
) -> Result<Response, ContractError> {
 ...
 ...
            .add_attribute("native_token", native_token.clone())
 ...
 ...
}
```

src/state.rs (Lines 6-12)

```
pub struct Config {
    pub owner: Addr,
    pub creator: Addr,
    pub fees_percentage: u64,
    pub fee_address: Addr,
    pub enabled: bool,
}
```

src/msg.rs (Line 8-11)

```
pub struct InstantiateMsg {
    pub owner: Addr,
    pub fee_address: Addr,
}
```

4.3.2 Recommendation

Either utilize the `native_token` parameter in the contract logic or remove it if it's not needed. If it's intended to be used, add a corresponding field to the `Config` struct and update relevant functions to use this value.

4.3.3 Customer Response

Native token option has been removed, contract logic works with CW20 tokens now.

4.4 Inconsistent Token Handling In `get_token_account`

The current implementation of `get_token_account` and `transfer_token_message` functions uses inconsistent parameter naming and handling for different token types (native and CW20), leading to potential confusion and errors.

Inconsistent terminology: For native tokens, "denom" (denomination) is used to refer to the token type (e.g., "uinjective", "ujuno"). For CW20 tokens, the contract address is required.
Parameter misuse: The denom parameter is used for both native token denomination and CW20 contract address. contract_addr is used as the address to check the balance, which is confusing for CW20 tokens.

4.4.1 Code location

src/util.rs

4.4.2 Recommendation

Safe Handling of Missing Data: Instead of unwrapping directly, we now use `match` to safely handle the possibility that `collateral_prices.get(&denom)` might not return a value. This prevents the function from panicking and instead allows it to return a specific error if the price data for a given collateral type is not found.

- Refactor the `get_token_amount` and `transfer_token_message` functions for clarity and consistency.
- Validate addresses before use

```
let example_addr = deps.api.addr_validate(&example_addr)?;
```

4.4.3 Customer Response

Native token option has been removed (see entry 4.3), contract logic works with CW20 tokens now.

4.5 Use Of Inline Static Values

The fee percentage (`fees_percentage`) is currently set as a static value and also not changable, which may limit the contract's flexibility.

4.5.1 Code location

src/msg.rs (Lines 9-13)

```
pub struct InstantiateMsg {
    pub owner: Addr,
    pub fee_address: Addr,
}
```

src/contract.rs (Lines 33-39)

```
let config = Config {
   // ...
   fees_percentage: 1u64,
   // ...
};
```

src/contract.rs (Lines 33-39)

```
pub fn query_config(deps: Deps) -> StdResult<ConfigResponse> {
   // ...
   Ok(ConfigResponse {
    // ...
    fees_percentage: 1u64,
   })
}
```

4.5.2 Recommendation

Implement dynamic fee percentage setting and retrieval, or disable this functionality.

- Update `query_config` to return the actual `fees_percentage` from the `Config`.
- Consider passing the value when initializing the `Config`.
- Consider adding a `fees_percentage` field to `InstantiateMsg`.
- Consider implementing a function to update `fees_percentage` if needed.

4.5.3 Customer Response

Fee percentage is now being set at contract initialization, and cannot be changed after this moment. This method was proposed and executed by the Kira team due to the demand to minimize the impact of admin being hacked. Only the `fee_address` field will be mutable in the contract, this setup won't affect LP users in a compromised admin scenario.

4.6 Inefficient ID Format And Storage

The current ID format is unnecessarily long and redundant, consuming excessive storage.

4.6.1 Code location

src/contract.rs (Lines 96-120)

```
let id = format!(
    "{}-{}-{:x}",
    owner.clone(),
    denom.clone(),
    env.block.time.seconds()
);
// ... (code omitted for brevity)
let lp = LiquidityPool {
    id: id.clone(),
    owner: owner.clone(),
    denom: denom.to_string().clone(),
    locktime: env.block.time.seconds() + locktime,
    amount: new_amount,
};
LP_MAP.save(deps.storage, id.clone(), &lp)?;
```

4.6.2 Recommendation

- Shorten the ID format, possibly using a counter or hash function.
- Remove redundant information from the `LiquidityPool` struct, as `owner` and `denom` are already part of the ID.
- Consider using a more efficient storage structure to reduce redundancy.

IRONNODE

ironnode.io

Thank you for choosing us!