SMART CONTRACT AUDIT REPORT

for

HurricaneSwap

Prepared By: Yiqun Chen

Hangzhou, China
July 23, 2021

Document Properties

Client	ALPHA VALUE LIMITED
Title	Smart Contract Audit Report
Target	HurricaneSwap
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 23, 2021	Shulin Bie	Final Release
1.0-rc	July 13, 2021	Shulin Bie	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the HurricaneSwap design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given branch of HurricaneSwap can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About HurricaneSwap

HurricaneSwap is the first cross-chain trading protocol based on Avalanche. The proprietary LP-bridge mechanism (Roke Protocol) enables Avalanche users to trade valuable assets from other public chains without leaving Avalanche. HurricaneSwap provides users with a high-performance and low-fees as well as unparalleled, seamless cross-chain trading experience.

The basic information of HurricaneSwap is as follows:

Table 1.1: Basic Information of HurricaneSwap

Item	Description
Target	HurricaneSwap
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 23, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Caijiawen/HurricaneSwap-contract.git (6cc5b77)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Caijiawen/HurricaneSwap-contract.git (7242a99)

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [13]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `HurricaneSwap` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	3	■ ■ ■
Low	3	■ ■ ■
Informational	3	■ ■ ■
Undetermined	0	
Total	10	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key HurricaneSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logics	Confirmed
PVE-002	Medium	Potential Reentrancy Risk in AvaxPool::deposit()/withdraw()	Time and State	Confirmed
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Confirmed
PVE-004	Informational	Recommended Explicit Pool Validity Checks	Security Features	Fixed
PVE-005	Medium	Logic Error Of addLiquidityAVAX()	Business Logics	Confirmed
PVE-006	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-007	Low	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-008	Low	Proper Event Usage In HurricaneSwap Implementation	Coding Practices	Fixed
PVE-009	Informational	Improper Logic Of _compareAmounts()	Coding Practices	Fixed
PVE-010	High	Logic Error Of Swaps For Deflationary Tokens	Business Logics	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: AvaxPool
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The HurricaneSwap protocol provides an incentive mechanism that rewards the staking of supported assets with the governance token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via add() and the weights of supported pools can be adjusted via set(). When analyzing the pool weight update routine set(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's hrn allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.1: AvaxPool::set()

If the call to massUpdatePools() is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

Status The issue has been confirmed by the team. The team decides to leave it and the third `_withUpdate` parameter will always be true.

3.2 Potential Reentrancy Risk in AvaxPool::deposit()/withdraw()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: `AvaxPool`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [6]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

In the `AvaxPool` contract, we notice both of the `deposit()` and `withdraw()` functions have potential reentrancy risk. In the following, we use the `deposit()` routine as an example. To elaborate, we show below the code snippet of the `deposit()` routine in `AvaxPool`. In the `depositHrnAndToken()` function, we notice `IERC20(multLpToken).safeTransfer(_user, tokenPending)` (line 325) will be called to transfer the accumulated rewards to the user before depositing new underlying assets into the `AvaxPool` and `pool.lpToken.safeTransferFrom(_user, address(this), _amount)` (line 329) will be called to transfer the underlying assets into the `AvaxPool`. If the `multLpToken` or `pool.lpToken` faithfully implements the ERC777-like standard, then the `deposit()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `IERC20(multLpToken).safeTransfer(_user, tokenPending)` (line 325) or `pool.lpToken.safeTransferFrom(_user, address(this), _amount)` (line 329) before the actual transfer of the underlying assets occurs. By doing so, we can effectively keep `user.rewardDebt` and `user.multLpRewardDebt` intact (used for the calculation of pending rewards at line 315 and line 323). With a lower `user.rewardDebt` and `user.multLpRewardDebt`, the re-entered `deposit()` is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains, just like earlier Uniswap/imBTC hack [15].

```
300     // Deposit LP tokens to AvaxPool for HRN allocation.
301     function deposit(uint256 _pid, uint256 _amount) public notPause {
302         PoolInfo storage pool = poolInfo[_pid];
303         if (isMultLP(address(pool.lpToken))) {
304             depositHrnAndToken(_pid, _amount, msg.sender);
305         } else {
306             depositHrn(_pid, _amount, msg.sender);
307         }
308     }
309 
310     function depositHrnAndToken(uint256 _pid, uint256 _amount, address _user) private {
311         PoolInfo storage pool = poolInfo[_pid];
312         UserInfo storage user = userInfo[_pid][_user];
313         updatePool(_pid);
314         if (user.amount > 0) {
315             uint256 pendingAmount = user.amount.mul(pool.accHrnPerShare).div(1e12).sub(
                    user.rewardDebt);
316             if (pendingAmount > 0) {
317                 safeHrnTransfer(_user, pendingAmount);
318             }
319             uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
320             IMasterChefAvax(multLpChef).deposit(poolCorrespond[_pid], 0);
321             uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
322             pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(
                    beforeToken).mul(1e12).div(pool.totalAmount));
323             uint256 tokenPending = user.amount.mul(pool.accMultLpPerShare).div(1e12).sub
                    (user.multLpRewardDebt);
324             if (tokenPending > 0) {
325                 IERC20(multLpToken).safeTransfer(_user, tokenPending);
326             }
327         }
328         if (_amount > 0) {
329             pool.lpToken.safeTransferFrom(_user, address(this), _amount);
330             if (pool.totalAmount == 0) {
```

```
                IMasterChefAvax(multLpChef).deposit(poolCorrespond[_pid], _amount);
                user.amount = user.amount.add(_amount);
                pool.totalAmount = pool.totalAmount.add(_amount);
            } else {
                uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
                IMasterChefAvax(multLpChef).deposit(poolCorrespond[_pid], _amount);
                uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
                pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(
                    beforeToken).mul(1e12).div(pool.totalAmount));
                user.amount = user.amount.add(_amount);
                pool.totalAmount = pool.totalAmount.add(_amount);
            }
        }
        user.rewardDebt = user.amount.mul(pool.accHrnPerShare).div(1e12);
        user.multLpRewardDebt = user.amount.mul(pool.accMultLpPerShare).div(1e12);
        emit Deposit(_user, _pid, _amount);
    }
```

Listing 3.2: `AvaxPool::deposit()`

Note the `withdraw()` routine in the same contract shares the same issue.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been confirmed by the team. The team decides to leave it as ERC777 tokens will not be used in the `HurricaneSwap` protocol.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AvaxPool`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.3: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `AvaxPool::addMultLP()` routine as an example. This routine is designed to approve a specific token for `multLpChef` contract. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 100): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
98   function addMultLP(address _addLP) public onlyOwner returns (bool) {
99       require(_addLP != address(0), "LP is the zero address");
100      IERC20(_addLP).approve(multLpChef, uint256(- 1));
101      return EnumerableSet.add(_multLP, _addLP);
102  }
```

Listing 3.4: `AvaxPool::addMultLP()`

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `AvaxPool::safeHrnTransfer()` routine in the `AvaxPool` contract. If the `USDT` token is supported as `hrn`, the unsafe version of `hrn.transfer(_to, hrnBal)` (line 461) and `hrn.`

`transfer(_to, _amount)` (line 463) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value). We may intend to replace `transfer()` with `safeTransfer()`.

```
457     // Safe HRN transfer function, just in case if rounding error causes pool to not
            have enough HRNs.
458     function safeHrnTransfer(address _to, uint256 _amount) internal {
459         uint256 hrnBal = hrn.balanceOf(address(this));
460         if (_amount > hrnBal) {
461             hrn.transfer(_to, hrnBal);
462         } else {
463             hrn.transfer(_to, _amount);
464         }
465     }
```

Listing 3.5: `AvaxPool::safeHrnTransfer()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()`.

Status The issue has been confirmed by the team. The team decides to leave it as non-compliant ERC20 tokens will not be used in the `HurricaneSwap` protocol.

3.4 Recommended Explicit Pool Validity Checks

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AvaxPool`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

The reward mechanism in `HurricaneSwap` has a central contract, i.e.,`AvaxPool`, that has been tasked with the reward distribution to various pools and stakers. In the following, we show the key `pool` data structure. Note all added pools are maintained in an array `poolInfo`.

```
    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool.
            Hurricanes to distribute per block.
        uint256 lastRewardBlock;  // Last block number that Hurricanes distribution
            occurs.
        uint256 accHrnPerShare; // Accumulated Hurricanes per share, times 1e12.
        uint256 accMultLpPerShare; //Accumulated multLp per share
```

```
        uint256 totalAmount;    // Total amount of current pool deposit.
    }

    // The hrn Token!
    IHRN public hrn;
    // hrn tokens created per block.
    uint256 public hrnPerBlock;
    // Info of each pool.
    PoolInfo[] public poolInfo;
```

Listing 3.6: AvaxPool.sol

When there is a need to add a new pool, set a new allocPoint for an existing pool, stake (by depositing the supported assets), unstake (by redeeming previously deposited assets), query pending rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range [0, poolInfo.length-1]. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say validatePool. This new modifier essentially ensures the given _pool_id or _pid indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not.

```
    function deposit(uint256 _pid, uint256 _amount) public notPause {
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            depositHrnAndToken(_pid, _amount, msg.sender);
        } else {
            depositHrn(_pid, _amount, msg.sender);
        }
    }

    function depositHrnAndToken(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        updatePool(_pid);
        ...
    }

    function depositHrn(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        updatePool(_pid);
        ...
    }
```

Listing 3.7: AvaxPool::deposit()&&depositHrnAndToken()&&depositHrn()

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including set(), deposit(), withdraw(), emergencyWithdraw(), pending() and updatePool

().

Recommendation Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
    modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
    }

    function deposit(uint256 _pid, uint256 _amount) public validatePool(_pid) notPause {
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            depositHrnAndToken(_pid, _amount, msg.sender);
        } else {
            depositHrn(_pid, _amount, msg.sender);
        }
    }
```

Listing 3.8: AvaxPool::deposit()

Status The issue has been addressed in this commit: `99def47`.

3.5 Logic Error Of addLiquidityAVAX()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AvaxHurricaneRouter`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

`HurricaneSwap` on `Avalanche` is designed as an evolutional improvement of `UniswapV2`, which is a major decentralized exchange (DEX) running on top of the `Ethereum` blockchain. `HurricaneSwap` follows `UniswapV2`'s core design, but extends with `restricted tokens` feature. If both of the tokens in the pool are restricted, only the privileged `owner` account can continue to add or remove liquidity for the pool.

To elaborate, we show below the related code snippet of the `AvaxHurricaneRouter` contract. Both of the `addLiquidity()` and `addLiquidityAVAX()` functions are used by liquidity providers to add liquidity for the pool. If one of the underlying assets that the user provides is `AVAX`, the `addLiquidityAVAX()` function will be used, otherwise the `addLiquidity()` function will be used. In the `addLiquidity()` function, we notice that only the privileged `owner` account can continue to add liquidity for the pool if both of the tokens in the pool are restricted (line 69 - line 77). However, it comes to our attention that the `addLiquidityAVAX()` function does not have the same logic. We may intend to keep consistency between the `addLiquidity()` and `addLiquidityAVAX()` functions.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external ensure(deadline) returns (uint amountA, uint amountB, uint liquidity) {
        {
            bool hasA = IHurricaneFactory(factory).restrictedTokens(tokenA);
            bool hasB = IHurricaneFactory(factory).restrictedTokens(tokenB);
            if (hasA && hasB) {
                (bool lpSwitch, bool swapSwitch) = IHurricaneFactory(factory).getSwitch
                    ();
                require(lpSwitch && !swapSwitch, 'Hurricane: liquidity closed or swap
                    open');
                require(msg.sender == IHurricaneFactory(factory).getOwner(), 'Hurricane:
                     not allowed');
            }
        }

        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = AvaxHurricaneLibrary.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IHurricanePair(pair).mint(to);

        //AddLiquidity(address pair, address tokenA, address tokenB, uint tokenADesired,
             uint tokenBDesired, uint tokenAmin, uint tokenBmin, uint lpAmount, string
            method);
        emit AddLiquidity(pair, tokenA, tokenB, amountADesired, amountBDesired,
            amountAMin, amountBMin, liquidity, 'addLiquidity');
        //emit Method('addLiquidity');
    }
```

Listing 3.9: `AvaxHurricaneRouter::addLiquidity()`

```
    function addLiquidityAVAX(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountAVAXMin,
        address to,
        uint deadline
    ) external payable returns (uint amountToken, uint amountAVAX, uint liquidity) {

        {
```

```
            require(deadline >= block.timestamp, 'HurricaneRouter: EXPIRED');
        }

        (amountToken, amountAVAX) = _addLiquidity(
            token,
            WAVAX,
            amountTokenDesired,
            msg.value,
            amountTokenMin,
            amountAVAXMin
        );
        address pair = AvaxHurricaneLibrary.pairFor(factory, token, WAVAX);
        TransferHelper.safeTransferFrom(token, msg.sender, pair, amountToken);
        IWAVAX(WAVAX).deposit{value : amountAVAX}();
        assert(IWAVAX(WAVAX).transfer(pair, amountAVAX));
        liquidity = IHurricanePair(pair).mint(to);
        // refund dust AVAX, if any
        if (msg.value > amountAVAX) TransferHelper.safeTransferAVAX(msg.sender, msg.
            value - amountAVAX);

        emit AddLiquidity(pair, token, WAVAX, amountTokenDesired, msg.value,
            amountTokenMin, amountAVAXMin, liquidity, 'addLiquidityAVAX');
        //emit Method('addLiquidityAVAX');
    }
```

Listing 3.10: `AvaxHurricaneRouter::addLiquidityAVAX()`

Recommendation Validate whether both of the tokens in the pool are restricted at the beginning of the `addLiquidityAVAX()` function. If so, only the privileged `owner` account can continue to add liquidity for the pool.

Status The issue has been confirmed by the team. The team decides to leave it as `AVAX` will never be restricted in the `HurricaneSwap` protocol.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `AvaxHurricaneRouter/HecoHurricaneRouter`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

In the `AvaxHurricaneRouter` contract, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. To elaborate, we show below the related code snippet of the contract. The `removeLiquidity()` function is used by the liquidity providers to remove liquidity from the pool. We notice the `AvaxHurricaneLibrary::pairFor()` is called at line 184 to calculate the pool address. However, the local `pair` variable storing the pool address is not used throughout the `removeLiquidity()` function. For better gas efficiency, we may intend to remove the redundant code.

```
156     function removeLiquidity(
157         address tokenA,
158         address tokenB,
159         uint liquidity,
160         uint amountAMin,
161         uint amountBMin,
162         address to,
163         uint deadline
164     ) public ensure(deadline) returns (uint amountA, uint amountB) {
165         ...
166         address pair = AvaxHurricaneLibrary.pairFor(factory, tokenA, tokenB);
167         (amountA, amountB) = _removeLiquidity(tokenA, tokenB, liquidity, amountAMin,
            amountBMin, to);
168         emit RemoveLiquidity(tokenA, tokenB, amountAMin, amountBMin, liquidity, '
            removeLiquidity');
169         //emit Method('removeLiquidity');

171     }
```

Listing 3.11: `AvaxHurricaneRouter::removeLiquidity()`

Note a number of routines can be similarly improved, including `AvaxHurricaneRouter::removeLiquidityAVAX()`, `HecoHurricaneRouter::removeLiquidity()` and `HecoHurricaneRouter::removeLiquidity-AVAX()`.

Recommendation Remove the redundant code from the above-mentioned functions.

Status The issue has been addressed by the following commit: `e6a5e9b`.

3.7 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

In the HurricaneSwap protocol, there is a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the owner account.

```
    function transferToken(
        address token,
        address to,
        uint256 amount
    ) external {
        {
            require(msg.sender == IHurricaneFactory(factory).getOwner(), 'HurricaneRouter: only owner');
        }
        if (token == token1  token == token0) {
        _safeTransfer(token, to, amount);
        }
    }
```

Listing 3.12: AvaxHurricanePair::transferToken()

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the owner is not governed by a DAO-like structure. Note that a compromised owner account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the HurricaneSwap design.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team.

3.8 Proper Event Usage In HurricaneSwap Implementation

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the dynamics of the ERC20Factory contract that is designed to act as a factory to create and manage ERC20 tokens for the HurricaneSwap protocol. we notice the emitted ERC20ChangeUser event (line 48) contains incorrect information.

To elaborate, we show below the related code snippet of the contract. Specifically, the event is defined as `event ERC20ChangeUser(address tokenAddress, address oldPauser, address newPauser, address oldOperator, address newOperator)`, we notice the second `oldPauser` parameter and the fourth `oldOperator` parameter indicate the old `_pauser` and `_operator` of the ERC20 token. However, in the `changeTokenUser()` function, it comes to our attention that `address old_pauser = token._pauser()` (line 46) and `address old_operator = token._operator()` (line 47) are behind `token.changeUser(new_operator, new_pauser)` (line 45). In essence, the local `old_pauser` and `old_operator` variables save the new `_pauser` and `_operator` of the ERC20 token, which results that the emitted `ERC20ChangeUser` event (line 48) contains incorrect information.

```
43     function changeTokenUser(address token_address, address new_pauser, address
           new_operator) onlySuperAdmin public {
44         ERC20Template token = ERC20Template(token_address);
45         token.changeUser(new_operator, new_pauser);
46         address old_pauser = token._pauser();
47         address old_operator = token._operator();
48         emit ERC20ChangeUser(token_address, old_pauser, new_pauser, old_operator,
               new_operator);
49     }
```

Listing 3.13: `ERC20Factory::changeTokenUser()`

Moreover, while examining the events that reflect the AvaxHurricaneFactory dynamics, we notice there is a lack of emitting an event to reflect owner changes. To elaborate, we show below the related code snippet of the contract.

```
    function setOwner(address _owner) external {
        require(msg.sender == owner, 'Hurricane: FORBIDDEN');
        owner = _owner;
    }
```

Listing 3.14: `AvaxHurricaneFactory::setOwner()`

With that, we suggest to add a new event `NewOwner` whenever the new `owner` is changed. Also, the new `owner` information is better `indexed`. Note each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the `owner` information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Note the `setOwner()` function in the `HecoHurricaneFactory` contract can be improved similarly.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed in this commit: `ba98e85`.

3.9 Improper Logic Of _compareAmounts()

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `HecoHurricaneRouterPart`
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [5]

Description

In the `HecoHurricaneRouterPart` contract, we observe the internal `_compareAmounts()` function is used by various functions in the contract. In the following, we use the `swapExactTokensForTokens()` routine as an example. To elaborate, we show below the related code snippet of the contract. In the `swapExactTokensForTokens()` function, the internal `_compareAmounts()` function is called (line 58) to validate whether the fifth input `realAmounts` array parameter of the function is equal to the local `amounts` array variable calculated by `HecoHurricaneLibrary.getAmountsOut(factory, amountIn, path)` (line 56), which saves the actual amounts of all the tokens specified by the third input `path` parameter over the period of this transaction.

In the `_compareAmounts()` function, if the length of the two input arrays are different, the function should return false immediately. However, we notice that the function will continue forward even though the length of the two input arrays are different (line 16 - line 18).

```
41     function swapExactTokensForTokens(
42         uint amountIn,
43         uint amountOutMin,
44         address[] calldata path,
45         address to,
46         uint[] calldata realAmounts,
47         uint deadline
48     ) external ensure(deadline) returns (uint[] memory amounts) {
```

```
        {
            require(msg.sender == IHurricaneFactory(factory).getOwner(), '
                HurricaneRouter: EXPIRED');
            (bool lpSwitch, bool swapSwitch) = IHurricaneFactory(factory).getSwitch();
            require(!lpSwitch && swapSwitch, 'Hurricane: not allowed');
        }

        amounts = HecoHurricaneLibrary.getAmountsOut(factory, amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'HurricaneRouter:
            INSUFFICIENT_OUTPUT_AMOUNT');
        require(_compareAmounts(realAmounts, amounts), 'HurricaneRouter: swap wrong
            order');
        ...
    }
```

Listing 3.15: `HecoHurricaneRouterPart::swapExactTokensForTokens()`

```
    function _compareAmounts(uint[] memory amountsIn, uint[] memory amountsOut) internal
         virtual pure returns (bool result) {
        result = true;
        if (amountsIn.length != amountsOut.length) {
            result = false;
        }
        for (uint i; i < amountsIn.length - 1; i++) {
            if (amountsIn[i] != amountsOut[i]) {
                result = false;
                break;
            }
        }
    }
```

Listing 3.16: `HecoHurricaneRouterPart::_compareAmounts()`

Recommendation The `_compareAmounts()` function should return false immediately if the length of the two input arrays are different.

Status The issue has been addressed in this commit: `99def47`.

3.10 Logic Error Of Swaps For Deflationary Tokens

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `HecoHurricaneRouterPart`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

In the `HecoHurricaneRouterPart` contract, we observe both of the `swapExactTokensForTokens()` and `swapExactTokensForTokensSupportingFeeOnTransferTokens()` functions are used to swap the exact amount of one token to another specified by the third input `path` parameter. Compared with the `swapExactTokens-ForTokens()` function, the `swapExactTokensForTokensSupportingFeeOnTransferTokens()` function supports the swap of deflationary tokens. While examining the logic of them, we observe an incorrect logic in the `swapExactTokensForTokensSupportingFeeOnTransferTokens()` function.

To elaborate, we show below the related code snippet of the contract. In the `swapExactTokens-ForTokens()` function, assume the third input `path` parameter is `[tokenA, tokenB, tokenC]`, the `amounts` variable calculated by `HecoHurricaneLibrary.getAmountsOut(factory, amountIn, path)` (line 56) will save the actual amount of tokenA, tokenB and tokenC. Only when the fifth input `realAmounts` parameter includes the actual amount of tokenA, tokenB and tokenC, the `swapExactTokensForTokens()` function will work well. Therefore, the fifth input `realAmounts` parameter will include the actual amount of all the tokens in the `path` in the real scenarios.

However, in the `swapExactTokensForTokensSupportingFeeOnTransferTokens()` function, if we assume the third input `path` parameter is `[tokenA, tokenB, tokenC]`, the `amounts` variable calculated by `_swapSupport-ingFeeOnTransferTokens(path, to)` (line 228) will only save the actual amount of tokenA and tokenB without tokenC. If the fifth input `realAmounts` parameter includes the actual amount of tokenA, tokenB and tokenC as the `swapExactTokensForTokens()` function, the `swapExactTokensForTokens-SupportingFeeOnTransferTokens()` function will always be reverted.

```
41     function swapExactTokensForTokens(
42         uint amountIn,
43         uint amountOutMin,
44         address[] calldata path,
45         address to,
46         uint[] calldata realAmounts,
47         uint deadline
48     ) external ensure(deadline) returns (uint[] memory amounts) {

50         {
51             require(msg.sender == IHurricaneFactory(factory).getOwner(), '
                   HurricaneRouter: EXPIRED');
52             (bool lpSwitch, bool swapSwitch) = IHurricaneFactory(factory).getSwitch();
53             require(!lpSwitch && swapSwitch, 'Hurricane: not allowed');
54         }

56         amounts = HecoHurricaneLibrary.getAmountsOut(factory, amountIn, path);
57         require(amounts[amounts.length - 1] >= amountOutMin, 'HurricaneRouter:
               INSUFFICIENT_OUTPUT_AMOUNT');
58         require(_compareAmounts(realAmounts, amounts), 'HurricaneRouter: swap wrong
               order');
59         TransferHelper.safeTransferFrom(
60             path[0], msg.sender, HecoHurricaneLibrary.pairFor(factory, path[0], path[1])
```

```
                , amounts[0]
        );
        _swap(amounts, path, to);
        //emit SwapPath(amounts, path, to, 'swapExactTokensForTokens');
        emit SwapPath(amounts, amountIn, amountOutMin, path, to, '
            swapExactTokensForTokens');
        //emit Method("swapExactTokensForTokens");
    }
```

Listing 3.17: `HecoHurricaneRouterPart::swapExactTokensForTokens()`

```
    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint[] calldata realAmounts,
        uint deadline
    ) external ensure(deadline) {

        {
            require(msg.sender == IHurricaneFactory(factory).getOwner(), '
                HurricaneRouter: EXPIRED');
            (bool lpSwitch, bool swapSwitch) = IHurricaneFactory(factory).getSwitch();
            require(!lpSwitch && swapSwitch, 'Hurricane: liquidity open or swap closed')
                ;
        }

        TransferHelper.safeTransferFrom(
            path[0], msg.sender, HecoHurricaneLibrary.pairFor(factory, path[0], path[1])
                , amountIn
        );
        uint balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
        uint[] memory amounts = _swapSupportingFeeOnTransferTokens(path, to);
        require(
            IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >=
                amountOutMin,
            'HurricaneRouter: INSUFFICIENT_OUTPUT_AMOUNT'
        );
        require(_compareAmounts(realAmounts, amounts), 'HurricaneRouter: swap wrong
            order');
        //emit SwapPath(amounts, path, to, '
            swapExactTokensForTokensSupportingFeeOnTransferTokens');
        emit SwapPath(amounts,  amountIn, amountOutMin, path, to,'
            swapExactTokensForTokensSupportingFeeOnTransferTokens');
        //emit Method("swapExactTokensForTokensSupportingFeeOnTransferTokens");
    }
```

Listing 3.18: `HecoHurricaneRouterPart::swapExactTokensForTokensSupportingFeeOnTransferTokens()`

Note a number of functions can be similarly improved, including `AvaxHurricaneRouterPart::swapExactTokensForTokensSupportingFeeOnTransferTokens()` and `AvaxHurricaneRouterPart::swapExact-`

`AVAXForTokensSupportingFeeOnTransferTokens()`.

Recommendation There is no need to use `require(_compareAmounts(realAmounts, amounts), 'HurricaneRouter: swap wrong order')` (line 58) to keep the swap order. It becomes optional to remove the related codes.

Status The issue has been addressed in this commit: `ba98e85`.

4 | Conclusion

In this audit, we have analyzed the `HurricaneSwap` design and implementation. `HurricaneSwap` is the first cross-chain trading protocol based on `Avalanche`. The proprietary LP-bridge mechanism (Roke Protocol) enables `Avalanche` users to trade valuable assets from other public chains without leaving `Avalanche`. `HurricaneSwap` provides users with a high-performance and low-fees as well as unparalleled, seamless cross-chain trading experience. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[8] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[9] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[10] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[11] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[12] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[13] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[14] PeckShield. PeckShield Inc. https://www.peckshield.com.

[15] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[16] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.