SMART CONTRACT AUDIT REPORT

for

HurricaneSwap V2

Prepared By: Yiqun Chen

PeckShield

October 1, 2021

Document Properties

Client	ALPHA VALUE LIMITED
Title	Smart Contract Audit Report
Target	HurricaneSwap V2
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	October 1, 2021	Xuxian Jiang	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 HurricaneSwapV2 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 HurricaneSwapV2 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 V2

HurricaneSwap is the first cross-chain trading protocol based on Avalanche. The proprietary LP-bridge mechanism (Roke) enables Avalanche users to trade valuable assets from other public chains without leaving Avalanche. Taking the advantage of Avalanche with sub-second transaction times and low fees, HurricaneSwap provides users with a high-performance, low slippage, low-cost and seamless cross-chain trading experience.

The basic information of HurricaneSwapV2 is as follows:

Table 1.1: Basic Information of HurricaneSwap V2

Item	Description
Target	HurricaneSwap V2
Website	https://hurricaneswap.com
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 1, 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-v2-contract.git (ebfef65)

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-v2-contract.git (TBD)

1.2 About PeckShield

PeckShield Inc. [11] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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

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

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 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) [9], 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 `HurricaneSwapV2` 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	0	
Medium	1	■
Low	3	■ ■ ■
Informational	1	■
Total	5	

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 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key HurricaneSwap V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistency Between Document And Implementation	Coding Practices	
PVE-002	Low	Revisited HcSwapAvaxERC20::_approve() Visibility	Coding Practices	
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	
PVE-004	Informational	Incompatibility With Deflationary ERC20 Tokens	Business Logic	
PVE-005	Low	Possible Sandwich-Based DoS For LP Addition/Removal	Time And State	

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 Inconsistency Between Document And Implementation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: HcSwapBSCPair, HcSwapAvaxPair
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The HurricaneSwapV2 protocol has two customized DEX engines for BSC and Avax respectively. The DEX engine is inspired from UniswapV2 with extensions for the cross-chain support and customized fee. While reviewing the customized fee support, we notice the inconsistency between the document and the implementation.

In particular, we use the HcSwapBSCPair contract as an example. The inconsistency comes from the protocol fee collection. If the protocol fee is collected at every trade, it may unnecessarily impose an additional gas cost on every trade. To avoid this, accumulated fees are collected only when liquidity is deposited or withdrawn. The contract computes the accumulated fees, and mints new liquidity tokens to the fee beneficiary, immediately before any tokens are minted or burned. To elaborate, we show below the related _mintFee() function.

```
    // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
    function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn)
         {
        address feeTo = IUniswapV2Factory(factory).feeTo();
        feeOn = feeTo != address(0);
        uint _kLast = kLast; // gas savings
        if (feeOn) {
            if (_kLast != 0) {
                uint rootK = Math.sqrt(uint(_reserve0).mul(_reserve1));
                uint rootKLast = Math.sqrt(_kLast);
                if (rootK > rootKLast) {
                    uint numerator = totalSupply.mul(rootK.sub(rootKLast));
```

```
                    uint denominator = (rootK.mul(3) / 2).add(rootKLast);
                    uint liquidity = numerator / denominator;
                    if (liquidity > 0) _mint(feeTo, liquidity);
                }
            }
        } else if (_kLast != 0) {
            kLast = 0;
        }
    }
```

Listing 3.1: `HcSwapBSCPair::_mintFee()`

It comes to our attention that the current protocol fee occupies the 40% or 2/5, instead of the mentioned "1/6th of the growth in sqrt(k)". Note that another contract `HcSwapAvaxPair` shares the same issue.

Recommendation Revise the above `_mintFee()` function to make it consistent on the percentage of protocol fee for collection.

Status

3.2 Revisited HcSwapAvaxERC20::_approve() Visibility

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HcSwapAvaxERC20`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

`Solidity` supports four types of visibility for functions and state variables: `external`, `public`, `internal`, and `private`. In particular, functions have to be specified as being `external`, `public`, `internal` or `private`. For state variables, `external` is not possible. While examining the `HcSwapAvaxERC20` contract, we notice it is inherited by another `HcToken` contract. With that, certain `private` functions may need to be revisited especially when they are used in the child contracts.

In the following, we show four example functions from the `HcSwapAvaxERC20` contract. It comes to our attention that three of them are defined as `internal`, while one, i.e., `_approve()`, is defined as `private`. Note that `private` functions and state variables are only visible for the contract they are defined in and not in derived contracts. Considering it is inherited by `HcToken`, we suggest to make the `_approve()` visibility to be consistent with others, i.e., redefining it to be `internal`, instead of current `private`.

```
    function _mint(address to, uint value) internal {
        totalSupply = totalSupply.add(value);
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(address(0), to, value);
    }

    function _burn(address from, uint value) internal {
        balanceOf[from] = balanceOf[from].sub(value);
        totalSupply = totalSupply.sub(value);
        emit Transfer(from, address(0), value);
    }

    function _approve(address owner, address spender, uint value) private {
        allowance[owner][spender] = value;
        emit Approval(owner, spender, value);
    }

    function _transfer(address from, address to, uint value) internal  {
        balanceOf[from] = balanceOf[from].sub(value);
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(from, to, value);
    }
```

Listing 3.2: Internal Functions in HcSwapAvaxERC20

Recommendation Revisit the HcSwapAvaxERC20::_approve() visibility to be internal.

Status

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the HurricaneSwapV2 protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., configuring various parameters and assigning the operator roles). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    // only for cross bridge
    function directlyMint(uint liquidity, address to) external onlyOwner lock {
```

```
        _mint(to,liquidity);
        _sync();
    }

    // only for cross bridge
    function directlyBurn(uint liquidity, address from, address to, uint amount0, uint
        amount1) external onlyOwner lock {
        _burn(from,liquidity);
        _safeTransfer(token0, to, amount0);
        _safeTransfer(token1, to, amount1);
        _sync();
    }

    // only for cross bridge
    function directlySync(uint256 amount0,uint256 amount1) external onlyOwner lock{
        address _token0 = token0; // gas savings
        address _token1 = token1; // gas savings
        _safeTransfer(_token0, msg.sender, IERC20(_token0).balanceOf(address(this)).sub(
            amount0));
        _safeTransfer(_token1, msg.sender, IERC20(_token1).balanceOf(address(this)).sub(
            amount1));
        require(amount0.mul(amount1) >= uint(reserve0).mul(reserve1),"HcSwap::
            directlySync K");
        _sync();
    }
```

Listing 3.3: `HcSwapBSCPair::directlyMint()/directlyBurn()/directlySync()`

Note that if the privileged `owner` account or the configured `operator` is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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

3.4 Incompatibility With Deflationary ERC20 Tokens

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: HcSwapV2Router02, HcSwapAvaxRouter
- Category: Business Logic [7]
- CWE subcategory: CWE-708 [4]

Description

In HurricaneSwapV2, the HcSwapAvaxRouter contract operates as the main entry for interaction with DEX users. In particular, one entry routine, i.e., addLiquidity(), accepts asset transfer-in and mints the corresponding LP tokens to represent the depositor's share in the DEX pool. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) whenNotPaused returns (uint amountA, uint
        amountB, uint liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IUniswapV2Pair(pair).mint(to);
    }
```

Listing 3.4: HcSwapAvaxRouter::addLiquidity()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. In particular, a new set of swap functions that can accommodate the transfer fee can be introduced. (Another mitigation is to regulate the set of ERC20 tokens that are permitted into HurricaneSwap V2 for trading.)

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status

3.5 Possible Sandwich-Based DoS For LP Addition/Removal

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HcSwapAvaxRouter`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, `HurricaneSwap` aims to enable seamless cross-chain trading. The cross-chain trading still enforces the normal slippage control with the user-supplied `amountAMin/amountBMin`. While examining the slippage control enforcement, we notice the cross-chain trading may be potentially sandwiched to block liquidity providers from providing or withdrawing their liquidity.

To elaborate, we show below the `onAddLPCrossTask()` function in `HcSwapAvaxRouter`. This routine essentially handles the request to supply cross-chain liquidity with the help of an internal handler `_addLiquidityNoRevert()`. However, this function may return an unsuccessful return result (line 190).

```
182    function onAddLPCrossTask(LPQueue.LPAction memory action) internal returns (address
           tokenA, address tokenB, uint amountA, uint amountB, uint liquidity,
           CrossActionStatus success){
```

```
        (address bscTokenA, address bscTokenB, uint amountADesired, uint amountBDesired,
            uint amountAMin, uint amountBMin, , uint deadline) = LPQueue.decodeAddLP(action
            .payload);
        (tokenA, tokenB) = _mappingBSCTokenToAvax(bscTokenA, bscTokenB);
        // require(deadline >= block.timestamp, 'HcSwapV2Router: CROSS_EXPIRED');
        if (deadline >= block.timestamp) {
            address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
            require(pair != address(0), "HcSwapV2Router::onRemoveLPCrossTask: NO_PAIR");

            (amountA, amountB, success) = _addLiquidityNoRevert(tokenA, tokenB,
                amountADesired, amountBDesired, amountAMin, amountBMin);
            if (success == CrossActionStatus.SUCCESS) {
                IHcToken(tokenA).superMint(pair, amountA);
                IHcToken(tokenB).superMint(pair, amountB);
                liquidity = IUniswapV2Pair(pair).mint(msg.sender);
            }
        } else {
            success = CrossActionStatus.CROSS_EXPIRED;
        }
        return (bscTokenA, bscTokenB, amountA, amountB, liquidity, success);
    }
```

Listing 3.5: HcSwapAvaxRouter::onAddLPCrossTask()

In particular, this is possible as the `onAddLPCrossTask()` transaction may be sandwiched (via MEV) by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss or brings a larger slippage that may fail the request to add or removal liquidity. Fortunately, this sandwiched attack may come with the inherent risk for the possible loss of attacker's funds. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of investors.

Status

4 | Conclusion

In this audit, we have analyzed the `HurricaneSwapV2` design and implementation. `HurricaneSwapV2` is the first cross-chain trading protocol based on `Avalanche`. The proprietary LP-bridge mechanism enables `Avalanche` users to trade valuable assets from other public chains without leaving `Avalanche`. `HurricaneSwapV2` 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[4] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

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

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

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

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

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

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

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