SMART CONTRACT AUDIT REPORT

for

HLND

Prepared By: Yiqun Chen

PeckShield

January 11, 2022

Document Properties

Client	Legends Never Die
Title	Smart Contract Audit Report
Target	HLND
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 11, 2022	Xiaotao Wu	Final Release
1.0-rc	December 12, 2021	Xiaotao Wu	Release Candidate #1

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 design document and related source code of the `HLND` protocol, 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 version of smart contracts 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 HLND

`HLND` is a protocol token that is designed to deliver benefits to holders. By combining the power of reflection tokens and diversification, `HLND` creates a lucrative opportunity for those who get in early and simply hold. The longer `HLND` is held, the more rewards are earned. Holders of `HLND` are rewarded with two entirely different tokens over time. By earning two separate tokens, users automatically diversify their earnings and successfully position themselves into investments for both the short and long term.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of HLND

Item	Description
Name	Legends Never Die
Website	https://www.legendsneverdie.ae/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 11, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that our audit only covers the `flatten/HLND.sol`, `flatten/HLNDReferral.sol`, and `flatten/TokenDividendTracker.sol` contracts.

- https://github.com/mygittab/Legends-Never-Die-HLND-And-HLNDReferral-SmartContract (ba179cd)

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

- https://github.com/mygittab/Legends-Never-Die-HLND-And-HLNDReferral-SmartContract (145cacd)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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 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) [11], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 design and implementation of the HLND protocol smart contracts. 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 logics, 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	2	■■
Low	3	■■■
Informational	0	
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 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 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Sandwich/MEV Attack In HLND	Time and State	Confirmed
PVE-002	Low	Improved Logic In HLND::setAddressParameters()	Business Logic	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Potential Sandwich/MEV Attack In HLND

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: HLND
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

While examining the `HLND` contract, we notice there are several functions that can be improved with slippage control. In the following, we take the `swapTokensForBNB()` routine as an example. To elaborate, we show below the related code snippet of the `HLND` contract. According to the design, the `swapTokensForBNB()` function is used to swap `HLND` to `BNB`. In the function, the `swapExactTokensForETHSupportingFeeOnTransferTokens()` function of `PancakeSwap` is called (line 2344) to swap the exact `HLND` amount to `BNB`. However, we observe the second input `amountOutMin` parameter is assigned to `0`, which means this transaction does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks.

```
2335     function swapTokensForBNB(uint256 tokenAmount) internal {
2336         // generate the uniswap pair path of token -> weth
2337         address[] memory path = new address[](2);
2338         path[0] = address(this);
2339         path[1] = addressParameters.router.WETH();

2341         _approve(address(this), address(addressParameters.router), tokenAmount);

2343         // make the swap
2344         addressParameters.router.swapExactTokensForETHSupportingFeeOnTransferTokens(
2345             tokenAmount,
2346             0, // accept any amount of ETH
2347             path,
2348             address(this),
2349             block.timestamp
```

```
        );
    }
```

Listing 3.1: `HLND::swapTokensForBNB()`

Note other routines such as `addLiquidity()`, `buyBackAndBurn()`, and `swapTokensForDividendToken()` in the same contract can be similarly improved.

Recommendation Improve the above-mentioned functions by adding necessary slippage control.

Status This issue has been confirmed.

3.2 Improved Logic In HLND::setAddressParameters()

- ID: PVE-002
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: `HLND`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `HLND` contract provides a privileged function for the `owner` to set various protocol-wide contract addresses. While examining the routine, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that the calling of this function will revert if there is no change for the `router` address. Specifically, the internal calling of `TokenDividendTracker::excludeFromDividends()` will revert since this `router` address has been excluded from dividends in the initialize phase (line 1316).

```
    /// @notice Set Address Parameters
    /// @param _newMarketingWallet new marketing wallet
    /// @param _newTeamWallet new team wallet
    /// @param _newCakeContract new cake contract
    /// @param _newAdaContract new ada contract
    /// @param _router new router
    /// @param _referral new referral

    function setAddressParameters(
        address _newMarketingWallet,
        address _newTeamWallet,
        address _newCakeContract,
        address _newAdaContract,
        IUniswapV2Router02 _router,
        IHLNDReferral _referral
    ) external onlyOwner {
```

```
        _excludeFromFees(_newMarketingWallet, true);
        _excludeFromFees(_newTeamWallet, true);
        addressParameters.teamWallet = _newMarketingWallet;
        addressParameters.marketingWallet = _newTeamWallet;
        addressParameters.cakeDividendToken = _newCakeContract;
        addressParameters.adaDividendToken = _newAdaContract;
        addressParameters.router = _router;
        addressParameters.referral = _referral;

        _excludeFromDividend(address(_router));
    }
```

Listing 3.2: `HLND::setAddressParameters()`

```
    /// @notice exclude from dividends
    /// @param account new address

    function excludeFromDividends(address account) external onlyAuthorizedOrOwner {
        require(!excludedFromDividends[account]);
        excludedFromDividends[account] = true;

        _setBalance(account, 0);
        tokenHoldersMap.remove(account);

        emit ExcludeFromDividends(account);
    }
```

Listing 3.3: `TokenDividendTracker::excludeFromDividends()`

Recommendation Add a new setter function for the `owner` to set the `router` address.

Status This issue has been fixed in the following commit: `145cacd`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DividendPayingToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [1]

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 `transfer()` routine and 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 transfer() routine. It is important to note that this routine 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.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.4: USDT Token Contract

In the following, we show the _withdrawDividendOfUser() routine in the DividendPayingToken contract. Suppose the token to be transferred is one of those non-compliant ERC20 tokens, the transfer () call (line 1078) will be reverted.

```
1070     /// @notice Withdraws the ether distributed to the sender.
1071     /// @dev It emits a `DividendWithdrawn` event if the amount of withdrawn ether is
             greater than 0.
1072     function _withdrawDividendOfUser(address user) internal returns (uint256) {
1073         uint256 _withdrawableDividend = withdrawableDividendOf(user);
1074
1075         if (_withdrawableDividend > minTokenBeforeSendDividend) {
1076             withdrawnDividends[user] = withdrawnDividends[user].add(
                     _withdrawableDividend);
1077             emit DividendWithdrawn(user, _withdrawableDividend);
1078             bool success = IERC20Upgradeable(dividendToken).transfer(user,
                     _withdrawableDividend);
1079
1080             if (!success) {
1081                 withdrawnDividends[user] = withdrawnDividends[user].sub(
                         _withdrawableDividend);
1082                 return 0;
1083             }
1084
```

```
            return _withdrawableDividend;
        }

        return 0;
    }
```

Listing 3.5: DividendPayingToken::_withdrawDividendOfUser()

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. To use this library you can add the following statement: using SafeERC20 for IERC20.

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

Status This issue has been fixed in the following commit: 5942f1d.

3.4 Improved Precision By Multiplication And Division Reordering

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: HLND
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (mul) and division (div) are involved.

In particular, we use the HLND::_transfer() as an example. This routine is called to transfer HLND token and will calculate the referer reward if certain conditions are met. And the referer reward amount is calculated with a combination of mul/div operations (line 2129). All these operations are intended for uint256. We point out that if there is a sequence of multiplication and division operations, it is always better to calculate the multiplication before the division (on the condition without introducing any extra overflows). By doing so, we can achieve better precision.

```
  function _transfer(
    address from,
    address to,
    uint256 amount
) internal override {
    require(from != address(0), "zero address");
    require(to != address(0), "zero address");
    require(boolParameters.tradingIsEnabled  (isExcludedFromFees[from]
        isExcludedFromFees[to]), "Trading not started");

    bool excludedAccount = isExcludedFromFees[from]  isExcludedFromFees[to];

    //check pair for referral, HLND->LND. from - pairs, to - user. User buy HLND for LND
        .
    if (
        boolParameters.tradingIsEnabled &&
        automatedMarketMakerPairsForReferral[from] &&
        !excludedAccount &&
        addressParameters.referral.isHashReferer(to)
    ) {
        uint256 refererReward = amount.div(DIVIDER).mul(addressParameters.referral.
            refererFee());
        address referer = addressParameters.referral.getReferer(to);
        uint256 contractBalanceReferer = IERC20Upgradeable(address(this)).balanceOf(
            referer);

        if (contractBalanceReferer + refererReward <= commonParameters.maxWalletToken) {
            _mint(referer, refererReward);
        }
    }

    if (boolParameters.tradingIsEnabled && automatedMarketMakerPairs[from] && !
        excludedAccount) {
        uint256 contractBalanceRecipient = balanceOf(to);
        require(contractBalanceRecipient + amount <= commonParameters.maxWalletToken, "
            Error amount");
    } else if (boolParameters.tradingIsEnabled && automatedMarketMakerPairs[to] && !
        excludedAccount) {
        require(amount <= commonParameters.maxSellTransactionAmount, "Error amount");

        uint256 contractTokenBalance = balanceOf(address(this));

        if (!boolParameters.swapping && contractTokenBalance >= commonParameters.
            swapTokensAtAmount) {
            boolParameters.swapping = true;

            if (boolParameters.marketingEnabled) {      //Evan: charge fee
                uint256 swapTokens = contractTokenBalance.div(swapParameters.totalFees).
                    mul(swapParameters.marketingFee);
```

```

                swapTokensForBNB(swapTokens);
                uint256 teamPortion = address(this).balance.div(10**2).mul(66);
                uint256 marketingPortion = address(this).balance.sub(teamPortion);
                transferToWallet(payable(addressParameters.marketingWallet),
                    marketingPortion);
                transferToWallet(payable(addressParameters.teamWallet), teamPortion);
            }

            if (boolParameters.buyBackAndLiquifyEnabled) {
                if (boolParameters.buyBackMode) {
                    swapTokensForBNB(contractTokenBalance.div(swapParameters.totalFees).
                        mul(swapParameters.buyBackAndLiquidityFee));
                } else {
                    swapAndLiquify(contractTokenBalance.div(swapParameters.totalFees).
                        mul(swapParameters.buyBackAndLiquidityFee));
                }
            }

            if (boolParameters.cakeDividendEnabled) {
                uint256 sellTokens = contractTokenBalance.div(swapParameters.totalFees).
                    mul(swapParameters.cakeDividendRewardsFee);

                sellTokens = setSellBalance(sellTokens);

                swapAndSendCakeDividends(sellTokens);
            }

            if (boolParameters.adaDividendEnabled) {
                uint256 sellTokens = contractTokenBalance.div(swapParameters.totalFees).
                    mul(swapParameters.adaDividendRewardsFee);
                sellTokens = setSellBalance(sellTokens);
                swapAndSendAdaDividends(sellTokens);
            }

            boolParameters.swapping = false;
        }

        if (!boolParameters.swapping && boolParameters.buyBackAndLiquifyEnabled &&
            boolParameters.buyBackMode) {
            uint256 buyBackBalanceBnb = address(this).balance;
            if (buyBackBalanceBnb >= commonParameters.minimumBalanceRequired && amount
                >= commonParameters.minimumSellOrderAmount) {
                boolParameters.swapping = true;

                if (buyBackBalanceBnb > commonParameters.buyBackUpperLimit) {
                    buyBackBalanceBnb = commonParameters.buyBackUpperLimit;
                }

                buyBackAndBurn(buyBackBalanceBnb.div(10**2));

                boolParameters.swapping = false;
```

```
            }
        }
    }

    if (boolParameters.tradingIsEnabled && !boolParameters.swapping && !excludedAccount)
         {
        uint256 fees = amount.div(DIVIDER).mul(swapParameters.totalFees);

        // if sell, multiply by swapParameters.sellFeeIncreaseFactor
        if (automatedMarketMakerPairs[to]) {
            fees = fees.div(DIVIDER).mul(swapParameters.sellFeeIncreaseFactor);
        }

        amount = amount.sub(fees);

        super._transfer(from, address(this), fees);
    }

    super._transfer(from, to, amount);

    try addressParameters.cakeDividendTracker.setBalance(from, balanceOf(from)) {} catch
         {}
    try addressParameters.adaDividendTracker.setBalance(from, balanceOf(from)) {} catch
        {}

    try addressParameters.cakeDividendTracker.setBalance(to, balanceOf(to)) {} catch {}
    try addressParameters.adaDividendTracker.setBalance(to, balanceOf(to)) {} catch {}

    if (!boolParameters.swapping) {
        uint256 gas = swapParameters.gasForProcessing;

        if (rand() <= swapParameters.cakeDividendPriority) {
            if (boolParameters.cakeDividendEnabled && boolParameters.sendCakeInTx) {
                try addressParameters.cakeDividendTracker.process(gas) returns (
                    uint256 iterations,
                    uint256 claims,
                    uint256 lastProcessedIndex
                ) {
                    emit ProcesseCakeDividendTracker(iterations, claims,
                        lastProcessedIndex, true, gas, tx.origin);
                } catch {}
            }

            if (boolParameters.adaDividendEnabled && boolParameters.sendAdaInTx) {
                try addressParameters.adaDividendTracker.process(gas) returns (
                    uint256 iterations,
                    uint256 claims,
                    uint256 lastProcessedIndex
                ) {
                    emit ProcessAdaDividendTracker(iterations, claims,
                        lastProcessedIndex, true, gas, tx.origin);
                } catch {}
```

```
2243             }
2244         } else {
2245             if (boolParameters.adaDividendEnabled && boolParameters.sendAdaInTx) {
2246                 try addressParameters.adaDividendTracker.process(gas) returns (
2247                     uint256 iterations,
2248                     uint256 claims,
2249                     uint256 lastProcessedIndex
2250                 ) {
2251                     emit ProcessAdaDividendTracker(iterations, claims,
                             lastProcessedIndex, true, gas, tx.origin);
2252                 } catch {}
2253             }
2254 
2255             if (boolParameters.cakeDividendEnabled && boolParameters.sendCakeInTx) {
2256                 try addressParameters.cakeDividendTracker.process(gas) returns (
2257                     uint256 iterations,
2258                     uint256 claims,
2259                     uint256 lastProcessedIndex
2260                 ) {
2261                     emit ProcesseCakeDividendTracker(iterations, claims,
                             lastProcessedIndex, true, gas, tx.origin);
2262                 } catch {}
2263             }
2264         }
2265     }
2266 }
```

Listing 3.6: `HLND::_transfer()`

Note similar issue also exists at a number of places in the same routine (e.g., lines 2150, 2153, 2161, 2163, 2168, 2176, 2201, and 2205).

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the following commit: `5942f1d`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `HLND` protocol, there is a certain privileged account, i.e., `owner`. When examining the related contracts, we notice inherent trust on this privileged account. To elaborate, we show below the related functions in the `HLND` contact.

Firstly, a number of `setters/updaters`, e.g., `setCommonParameters()`, `setBoolParameters()`, `setCakeDividendPriority()`, `setAddressParameters()`, `setSellTransactionMultiplier()`, `setBuyBackAndLiquifyEnabled()`, `setCakeDividendEnabled()`, `setAdaDividendEnabled()`, `setMarketingEnabled()`, `updateCakeDividendTracker()`, `updateAdaDividendTracker()`, `updateCakeDividendRewardFee()`, `updateAdaDividendRewardFee()`, `updateMarketingFee()`, `updateBuyBackAndLiquidityFee()`, `setAutomatedM-arketMakerPair()`, `setAutomatedMarketMakerPairForReferral()`, and `updateGasForProcessing()` allow for the `owner` to set/update various protocol-wide risk parameters for the `HLND` contract.

```
/// @notice Set common parameters
/// @param _maxTxnBuyTransaction max buy transaction
/// @param _maxTxnSellTransaction max sell transaction
/// @param _maxToken max token wallet
/// @param _swapAmount max swap amount
/// @param _minimumBalanceRequired minimum balance required
/// @param _minimumSellOrderAmount minimum sell order
/// @param _buyBackUpperLimit buy back upper limit

function setCommonParameters(
    uint256 _maxTxnBuyTransaction,
    uint256 _maxTxnSellTransaction,
    uint256 _maxToken,
    uint256 _swapAmount,
    uint256 _minimumBalanceRequired,
    uint256 _minimumSellOrderAmount,
    uint256 _buyBackUpperLimit
) external onlyOwner {
    commonParameters.maxBuyTransactionAmount = _maxTxnBuyTransaction * (10**18);
    commonParameters.maxSellTransactionAmount = _maxTxnSellTransaction * (10**18);
    commonParameters.maxWalletToken = _maxToken * (10**18);
    commonParameters.swapTokensAtAmount = _swapAmount * (10**18);
    commonParameters.minimumBalanceRequired = _minimumBalanceRequired;
    commonParameters.minimumSellOrderAmount = _minimumSellOrderAmount;
    commonParameters.buyBackUpperLimit = _buyBackUpperLimit;
}

/// @notice Set bool parameters
/// @param _newStatusSendCakeInTx send cake in swap tx
/// @param _newStatusAdaInTx send ada in swap tx
/// @param _tradingEnabled enable trading operations
/// @param _buyBackEnabled enable buyback mode
function setBoolParameters(
    bool _newStatusSendCakeInTx,
    bool _newStatusAdaInTx,
    bool _tradingEnabled,
    bool _buyBackEnabled
) external onlyOwner {
    boolParameters.sendCakeInTx = _newStatusSendCakeInTx;
    boolParameters.sendAdaInTx = _newStatusAdaInTx;
    boolParameters.tradingIsEnabled = _tradingEnabled;
```

```
        boolParameters.buyBackMode = _buyBackEnabled;
    }

    /// @notice cake priority
    /// @param _newAmount new value

    function setCakeDividendPriority(uint256 _newAmount) external onlyOwner {
        require(_newAmount >= 0 && _newAmount <= 100, "Error amount");
        swapParameters.cakeDividendPriority = _newAmount;
    }

    /// @notice Set Address Parameters
    /// @param _newMarketingWallet new marketing wallet
    /// @param _newTeamWallet new team wallet
    /// @param _newCakeContract new cake contract
    /// @param _newAdaContract new ada contract
    /// @param _router new router
    /// @param _referral new referral

    function setAddressParameters(
        address _newMarketingWallet,
        address _newTeamWallet,
        address _newCakeContract,
        address _newAdaContract,
        IUniswapV2Router02 _router,
        IHLNDReferral _referral
    ) external onlyOwner {
        _excludeFromFees(_newMarketingWallet, true);
        _excludeFromFees(_newTeamWallet, true);
        addressParameters.teamWallet = _newMarketingWallet;
        addressParameters.marketingWallet = _newTeamWallet;
        addressParameters.cakeDividendToken = _newCakeContract;
        addressParameters.adaDividendToken = _newAdaContract;
        addressParameters.router = _router;
        addressParameters.referral = _referral;

        _excludeFromDividend(address(_router));
    }

    /// @notice Set new Transaction Multiplier
    /// @param _multiplier multiplier

    function setSellTransactionMultiplier(uint256 _multiplier) external onlyOwner {
        swapParameters.sellFeeIncreaseFactor = _multiplier;
    }

    /// @notice Set Buy back and liquify enabled
    /// @param _enabled flag

    function setBuyBackAndLiquifyEnabled(bool _enabled) external onlyOwner {
        if (_enabled == false) {
            boolParameters.buyBackAndLiquifyEnabled = _enabled;
```

```
    } else {
        swapParameters.totalFees = swapParameters
            .buyBackAndLiquidityFee
            .add(swapParameters.marketingFee)
            .add(swapParameters.adaDividendRewardsFee)
            .add(swapParameters.cakeDividendRewardsFee);
        boolParameters.buyBackAndLiquifyEnabled = _enabled;
    }
}

/// @notice Set Cake Ddividend enabled
/// @param _enabled flag

function setCakeDividendEnabled(bool _enabled) external onlyOwner {
    if (_enabled == false) {
        boolParameters.cakeDividendEnabled = _enabled;
    } else {
        swapParameters.totalFees = swapParameters
            .cakeDividendRewardsFee
            .add(swapParameters.marketingFee)
            .add(swapParameters.adaDividendRewardsFee)
            .add(swapParameters.buyBackAndLiquidityFee);
        boolParameters.cakeDividendEnabled = _enabled;
    }
}

/// @notice Set Ada dividend enabled
/// @param _enabled flag

function setAdaDividendEnabled(bool _enabled) external onlyOwner {
    if (_enabled == false) {
        boolParameters.adaDividendEnabled = _enabled;
    } else {
        swapParameters.totalFees = swapParameters
            .adaDividendRewardsFee
            .add(swapParameters.marketingFee)
            .add(swapParameters.cakeDividendRewardsFee)
            .add(swapParameters.buyBackAndLiquidityFee);
        boolParameters.adaDividendEnabled = _enabled;
    }
}

/// @notice Set marketing enabled
/// @param _enabled flag

function setMarketingEnabled(bool _enabled) external onlyOwner {
    if (_enabled == false) {
        boolParameters.marketingEnabled = _enabled;
    } else {
        swapParameters.totalFees = swapParameters
            .marketingFee
            .add(swapParameters.adaDividendRewardsFee)
```

```
                .add(swapParameters.cakeDividendRewardsFee)
                .add(swapParameters.buyBackAndLiquidityFee);
            boolParameters.marketingEnabled = _enabled;
        }
    }
```

Listing 3.7: HLND::setters

```
    /// @notice Set Automated Market Maker Pair
    /// @param _pair address
    /// @param _value bool

    function setAutomatedMarketMakerPair(address _pair, bool _value) public onlyOwner {
        _setAutomatedMarketMakerPair(_pair, _value);
    }
```

Listing 3.8: HLND::setAutomatedMarketMakerPair()

```
    /// @notice Set Automated Market Maker Pair for Referral
    /// @param  _pair address
    /// @param  _value bool

    function setAutomatedMarketMakerPairForReferral(address _pair, bool _value) public
        onlyOwner {
        _setAutomatedMarketMakerPairForReferral(_pair, _value);
    }
```

Listing 3.9: HLND::setAutomatedMarketMakerPairForReferral()

```
    /// @notice Update gas for Processing Dividends
    /// @param  _newValue new value

    function updateGasForProcessing(uint256 _newValue) external onlyOwner {
        swapParameters.gasForProcessing = _newValue;
        emit GasForProcessingUpdated(_newValue, swapParameters.gasForProcessing);
    }
```

Listing 3.10: HLND::updateGasForProcessing()

Secondly, the `owner` of the `HLND` contract can exclude certain accounts from fees and dividends.

```
    /// @notice exclude some address from dividends
    /// @param _partnerOrExchangeAddress address of pair
    function prepareForPartnerOrExchangeListing(address _partnerOrExchangeAddress)
        external onlyOwner {
        addressParameters.cakeDividendTracker.excludeFromDividends(
            _partnerOrExchangeAddress);
        addressParameters.adaDividendTracker.excludeFromDividends(
            _partnerOrExchangeAddress);
        _excludeFromFees(_partnerOrExchangeAddress, true);
    }
```

Listing 3.11: HLND::prepareForPartnerOrExchangeListing()

```
/// @notice Exclude from Fees
/// @param _account address
/// @param _excluded bool

function excludeFromFees(address _account, bool _excluded) external onlyOwner {
    _excludeFromFees(_account, _excluded);
}
```

Listing 3.12: `HLND::excludeFromFees()`

```
/// @notice Exclude from Dividend
/// @param _account address

function excludeFromDividend(address _account) external onlyOwner {
    _excludeFromDividend(_account);
}
```

Listing 3.13: `HLND::excludeFromDividend()`

Thirdly, the `owner` of the `HLND` contract can manual buy back and burn HLND tokens and can withdraw `ERC20` tokens from the `HLND` contract.

```
/// @notice manual buy back and burn
/// @param _amount new value

function manualBuyBackAndBurn(uint256 _amount) public onlyOwner {
    uint256 balance = address(this).balance;
    require(boolParameters.buyBackAndLiquifyEnabled, "not enabled");
    require(balance >= commonParameters.minimumBalanceRequired.add(_amount), "amount
         is too big");

    if (!boolParameters.swapping) {
        buyBackAndBurn(_amount);
    }
}
```

Listing 3.14: `HLND::manualBuyBackAndBurn()`

```
/// @notice withdraw tokens from contract
/// @param _token address
/// @param _recipient address

function withdrawTokens(address _token, address _recipient) external onlyOwner {
    require(_token != address(0x0), "HLND: address is zero");
    require(_recipient != address(0x0), "HLND: address is zero");

    if (IERC20Upgradeable(_token).balanceOf(address(this)) > 0) {
        IERC20Upgradeable(_token).safeTransfer(_recipient, ERC20Upgradeable(_token).
            balanceOf(address(this)));
    }
}
```

Listing 3.15: `HLND::withdrawTokens()`

Lastly, the `owner` of the `HLND` contract can authorize certain accounts. Only these authorized accounts can call some certain privileged functions.

```
    /// @notice authorize contract for operation
    /// @param _account address
    /// @param _isAuthorized bool

    function authorize(address _account, bool _isAuthorized) external onlyOwner {
        authorized[_account] = _isAuthorized;
    }
```

Listing 3.16: `HLND::authorize()`

```
    function processDividendTracker(uint256 gas) external onlyAuthorized {
        (uint256 cakeIterations, uint256 cakeClaims, uint256 cakeLastProcessedIndex) =
            addressParameters.cakeDividendTracker.process(gas);
        emit ProcesseCakeDividendTracker(cakeIterations, cakeClaims,
            cakeLastProcessedIndex, false, gas, tx.origin);

        (uint256 adaIterations, uint256 adaClaims, uint256 adaLastProcessedIndex) =
            addressParameters.adaDividendTracker.process(gas);
        emit ProcessAdaDividendTracker(adaIterations, adaClaims, adaLastProcessedIndex,
            false, gas, tx.origin);
    }
```

Listing 3.17: `HLND::processDividendTracker()`

Note this trust issue also exists in the `HLNDReferral` and `TokenDividendTracker` contracts. We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `owner` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `owner` explicit to `HLND` users.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the HLND protocol. HLND is an innovative token that can deliver many benefits to holders. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

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

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

[10] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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