SMART CONTRACT AUDIT REPORT

for

HODL

Prepared By: Xiaomi Huang

PeckShield

November 27, 2024

Document Properties

Client	HODL
Title	Smart Contract Audit Report
Target	HODL
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 27, 2024	Xuxian Jiang	Final Release
1.0-rc	November 22, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
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 HODL 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 HODL

HODL is the first HODL to EARN memecoin, which allows users to buy any amount of HODL tokens to earn 10x HODL vesting tokens over a period of 10 years. In the meantime, it is designed to burn remaining vesting tokens if any HODL token is being sold. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The HODL

Item	Description
Name	HODL
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 27, 2024

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

- https://github.com/TechUpGroup/HODL_SMC.git (ae0a1be)

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

- https://github.com/TechUpGroup/HODL_SMC.git (e2e858e)

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

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 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) [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 implementation of the HODL protocol. 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	1	■
Low	3	■ ■ ■
Informational	0	
Total	4	

We have so far identified a list of potential issues. 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 and 3 low-severity vulnerabilities.

Table 2.1: Key HODL Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Front-Running/MEV For Reduced Return	Time And State	Confirmed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Low	Possible Griefing During Reward Withdrawal	Coding Practices	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Potential Sandwich/MEV For Reduced Returns

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Pool
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The audited protocol has a core Pool contract that is designed to interact with UniswapV3 DEX engine and manage the DEX liquidity. With that, it has the natural need of swapping tokens. Our analysis shows the token-swapping logic can be improved for better slippage control.

```
    function _swapTokensForEth(uint256 tokenAmount) private {
        // generate the uniswap pair path of token -> weth
        address[] memory path = new address[](2);
        path[0] = address(rewardToken);
        path[1] = routerV2.WETH();

        rewardToken.approve(address(routerV2), tokenAmount);

        // make the swap
        routerV2.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0, // accept any amount of ETH
            path,
            address(this),
            block.timestamp
        );
    }
```

Listing 3.1: Pool::_swapTokensForEth()

Specifically, if we examine the above _swapTokensForEth() implementation, the helper converts the token from rewardToken to ETH but uses 0 as amountOutMin. As a result, the router routerV2 performs the swap without the slippage control mechanism utilized. In other words, it may be sandwiched by a MEV bot for profit.

This is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched 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 and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. 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 front-running attack to better protect the interests of farming users. The same issue is also applicable to another routine, i.e., `_swapEthForTokens()`.

Status This issue has been confirmed.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [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 `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 3.2: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , 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 `transfer()/transferFrom()` as well, i.e., `safeTransfer()/safeTransferFrom()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.3: `SafeERC20::safeApprove()`

In current implementation, if we examine the `Pool::_addLiquidityV3()` routine that is designed to add liquidity. To accommodate the specific idiosyncrasy, there is a need to make use of `safeApprove()` twice: the first one resets the allowance while the second one sets the intended allowance (lines 222

248, and 278). In addition, the related `transferFrom()` routine, if any, is suggested to replace with `safeTransferFrom()`.

```
213     function _addLiquidityV3(address user, uint256 tokenAmount, uint256 ethAmount)
            private returns (
214         uint256 tokenId,
215         uint128 liquidity,
216         uint256 amount0,
217         uint256 amount1,
218         address pair
219     ) {
220         IWETH9 WETH = IWETH9(routerV2.WETH());
221         WETH.deposit{value: ethAmount}();
222         rewardToken.approve(address(nonfungiblePositionManager), tokenAmount);
223         WETH.approve(address(nonfungiblePositionManager), ethAmount);
224         if (address(rewardToken) < address(WETH)) {
225             INonfungiblePositionManager.MintParams memory params =
226             INonfungiblePositionManager.MintParams({
227                 token0: address(rewardToken),
228                 token1: address(WETH),
229                 fee: gFee,
230                 tickLower: (TickMath.MIN_TICK / TICK_SPACING) * TICK_SPACING,
231                 tickUpper: (TickMath.MAX_TICK / TICK_SPACING) * TICK_SPACING,
232                 amount0Desired: tokenAmount,
233                 amount1Desired: ethAmount,
234                 amount0Min: 0,
235                 amount1Min: 0,
236                 recipient: address(this),
237                 deadline: block.timestamp
238             });
239
240             (tokenId, liquidity, amount0, amount1) = nonfungiblePositionManager.mint(
                    params);
241
242             if (amount0 < tokenAmount) {
243                 rewardToken.approve(address(nonfungiblePositionManager), 0);
244                 uint256 refund0 = tokenAmount - amount0;
245                 rewardToken.safeTransfer(user, refund0);
246             }
247             ...
248         }
249         ...
250     }
```

Listing 3.4: `Pool::_addLiquidityV3()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note this issue affects all current leveraged strategies.

Status The issue has been resolved by following the above recommendation.

3.3 Possible Griefing During Reward Withdrawal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To incentivize protocol users, the `HODL` protocol will provide users with respective rewards. While examining the reward-claiming logic, we notice a griefing issue in current implementation.

In the following, we show the implementation of the related `claim()` routine. As the name indicates, this routine is designed to claim the reward. It has a signature verification procedure to ensure the reward is intended and legitimate and signed by the `TRUSTED_PARTY` account. It comes to our attention that a listening bot may notice the pending transaction to claim the reward and explicitly frontrun with the use of the same claim message to invalidate the nonce used in the pending transaction. By doing so, the claim transaction will be reverted.

```
    function claim(
        bytes32 nonce,
        uint256 amount,
        uint256 signTime,
        bytes memory signature
    ) external whenNotPaused nonReentrant {
        require(amount != 0, "Zero amount");
        require(amount <= rewardToken.balanceOf(address(this)), "Exceed fund");
        verifier.verifyClaim(nonce, msg.sender, amount, signTime, signature);
        rewardToken.safeTransfer(msg.sender, amount);
        emit Claimed(msg.sender, amount, nonce);
    }
```

Listing 3.5: `Pool::claim()`

Recommendation Revisit the above routine to add necessary caller binding with the signed message so that the claim logic will not be reverted.

Status This issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the HODL protocol, there is a privileged administrative account, i.e., owner. The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the Pool contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setVerifier(address _verifier) external onlyOwner {
        verifier = ISignatureVerifier(_verifier);
    }

    function setRouterV2(address _routerV2) external onlyOwner {
        routerV2 = IUniswapV2Router02(_routerV2);
    }

    function setNonfungiblePositionManager(address _nonfungiblePositionManager) external
         onlyOwner {
        nonfungiblePositionManager = INonfungiblePositionManager(
            _nonfungiblePositionManager);
    }

    function setOperator(address _operator) external onlyOwner {
        operator = _operator;
    }

    function setDeadAddress(address _dead) external onlyOwner {
        deadAddress = _dead;
    }

    function setTickSpacing(int24 _tickSpacing) external onlyOwner {
        require(_tickSpacing != 0, "not zero");
        TICK_SPACING = _tickSpacing;
    }

    function setGFee(uint24 _gFee) external onlyOwner {
        require(_gFee < 1e6, "exceed gFee");
        gFee = _gFee;
    }

    function pause() external onlyOwner {
```

```
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }
```

Listing 3.6: Example Privileged Operations in `Pool`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. Note that 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.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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 mitigated with the use of multi-sig to act as the owner account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `HODL` protocol, which is the first `HODL to EARN` memecoin and allows users to buy any amount of `HODL` tokens to earn `10x HODL` vesting tokens over a period of 10 years. In the meantime, it is designed to burn remaining vesting tokens if any `HODL` token is being sold. 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-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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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.