SMART CONTRACT AUDIT REPORT

for

HONEYFARM

Prepared By: Yiqun Chen

PeckShield

October 16, 2021

Document Properties

Client	HoneyFarm
Title	Smart Contract Audit Report
Target	HoneyFarm
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 16, 2021	Xuxian Jiang	Final Release
1.0-rc1	October 8, 2021	Xuxian Jiang	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 smart contract source code of the `HoneyFarm` 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 HoneyFarm

The basic information of HoneyFarm is as follows:

Table 1.1: Basic Information of HoneyFarm

Item	Description
Name	HoneyFarm
Website	https://honeyfarm.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 16, 2021

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

- https://github.com/RenovJ/honeyfarm-contracts.git (175b4a7)

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

- https://github.com/RenovJ/honeyfarm-contracts.git (1e7c1f5)

1.2 About PeckShield

PeckShield Inc. [5] 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 the OWASP Risk Rating Methodology [4]:

- 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 checklist of 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 Audit Checklist

Category	Checklist Items
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) [3], 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 Logic	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 HoneyFarm 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 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	2	
Low	2	
Informational	0	
Total	4	

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 2 low-severity vulnerabilities.

Table 2.1: Key HoneyFarm Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Missing Definition of onlyOwner in HoneyToken	Business Logic	Fixed
PVE-002	Low	Duplicate Pool/Strategy Detection and Prevention	Business Logic	Confirmed
PVE-003	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Confirmed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed

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

3 | Detailed Results

3.1 Missing Definition of onlyOwner in HoneyToken

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HoneyToken`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

The HoneyFarm protocol provides has the protocol/governance token `HoneyToken` that will be disseminated to community and protocol users. While examining the `HoneyToken` contract, we notice the use of a specific modifier `onlyOwner`, which is not defined yet.

To elaborate, we show below the related `mint()` function, which is designed to allow the privileged `owner` to mint additional tokens into circulation. However, it comes to our attention that this modifier is not defined.

```
contract HoneyToken is ERC20 {
    uint16 public transferTaxRate = 300;
    uint16 public constant MAXIMUM_TRANSFER_TAX_RATE = 1000;
    address public constant BURN_ADDRESS = 0x000000000000000000000000000000000000dEaD;
    address private _operator;
    event OperatorTransferred(address indexed previousOperator, address indexed newOperator);
    event TransferTaxRateUpdated(address indexed operator, uint256 previousRate, uint256 newRate);

    modifier onlyOperator() {
        require(_operator == msg.sender, "operator: caller is not the operator");
        _;
    }

    constructor() public ERC20("Honey token", "HONEY") {
        _operator = _msgSender();
        emit OperatorTransferred(address(0), _operator);
```

```
    }

    function mint(address _to, uint256 _amount) public onlyOwner {
        _mint(_to, _amount);
    }
    ...
}
```

Listing 3.1: The HoneyToken Contract

Recommendation Define the missing `onlyOwner` or inherit the `HoneyToken` from the `Ownable` contract.

Status The issue has been fixed in this commit: `1e7c1f5`.

3.2 Duplicate Pool/Strategy Detection and Prevention

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `YetiMaster`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

The HoneyFarm protocol provides an incentive mechanism that rewards the staking of supported assets with the governance token (e.g., `HoneyToken`). The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint *100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded governance tokens and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function add(
        uint256 _allocPoint,
        IERC20 _want,
        bool _withUpdate,
        address _strat,
        uint16 _depositFeeBP,
        bool _isWithdrawFee
    ) public onlyOwner {
        require(_depositFeeBP <= MAX_DEPOSIT_FEE_BP, "add: invalid deposit fee basis
            points");
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                want: _want,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accEarningsPerShare: 0,
                strat: _strat,
                depositFeeBP : _depositFeeBP,
                isWithdrawFee: _isWithdrawFee
            })
        );
    }
```

Listing 3.2: YetiMaster::add()

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate. We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers. Moreover, it is also applicable to validate the paired strategy is not duplicated!

Status The issue has been confirmed.

3.3 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `YetiMaster`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

As mentioned in Section 3.2, the HoneyFarm protocol provides an incentive mechanism that rewards the staking of supported assets with the governance token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

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

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate,
        uint16 _depositFeeBP,
        bool _isWithdrawFee
    ) public onlyOwner poolExists(_pid) {
        require(_depositFeeBP <= MAX_DEPOSIT_FEE_BP, "set: invalid deposit fee basis
            points");
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
        poolInfo[_pid].depositFeeBP = _depositFeeBP;
        poolInfo[_pid].isWithdrawFee = _isWithdrawFee;
    }
```

Listing 3.3: YetiMaster::set()

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

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

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate,
        uint16 _depositFeeBP,
        bool _isWithdrawFee
    ) public onlyOwner poolExists(_pid) {
        require(_depositFeeBP <= MAX_DEPOSIT_FEE_BP, "set: invalid deposit fee basis
            points");
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
        poolInfo[_pid].depositFeeBP = _depositFeeBP;
        poolInfo[_pid].isWithdrawFee = _isWithdrawFee;
    }
```

Listing 3.4: Revised YetiMaster::set()

Status The issue has been confirmed.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `YetiMaster`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event.*

The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.5: ZRX.sol

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

In the following, we show the `safeEarningsTransfer()` routine in the `YetiMaster` contract. If the `USDT` token is supported as `earningToken`, the unsafe version of `IERC20(earningToken).transfer(_to, EarningsBal)` (lines 380 and 382) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
    function safeEarningsTransfer(address _to, uint256 _EarningsAmt) internal {
        uint256 EarningsBal = IERC20(earningToken).balanceOf(address(this));
        if (_EarningsAmt > EarningsBal) {
            IERC20(earningToken).transfer(_to, EarningsBal);
        } else {
            IERC20(earningToken).transfer(_to, _EarningsAmt);
        }
    }
```

Listing 3.6: `YetiMaster::safeEarningsTransfer()`

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

Status The issue has been fixed in this commit: `1e7c1f5`.

4 | Conclusion

In this audit, we have analyzed the HoneyFarm design and implementation. The system presents a unique, robust offering as a mean of launching a new token and rewarding users for staking other LP or ERC20 tokens. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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