SMART CONTRACT AUDIT REPORT

for

EARNING.FARM

Prepared By: Shuxiao Wang

PeckShield

February 24, 2021

Document Properties

Client	earning.farm
Title	Smart Contract Audit Report
Target	earning.farm
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 24, 2021	Xudong Shao	Final Release
1.0-rc	February 23, 2021	Xudong Shao	Release Candidate
0.2	February 20, 2021	Xudong Shao	Add More Findings #1
0.1	February 19, 2021	Xudong Shao	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **earning.farm Protocol** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given 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 earning.farm Protocol

The earning.farm protocol is a is a yield seeking protocol that pools together users' assets and deposits them into other Defi protocols to seek high yield, while mitigate potential loss by only invest asset in homogeneity pools like stablecoin pools, BTC ERC20 token pools and ETH pools.

The basic information of the earning.farm protocol is as follows:

Table 1.1: Basic Information of earning.farm

Item	Description
Name	earning.farm
Website	https://earning.farm/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 24, 2021

In the following, we show the md5 hash value of the compressed file used in this audit:

- MD5 (cff-2021-02-17.zip) = fa5cbce61731f8c5b13c20c7a74eccc8

And here is the final md5 hash value of the patch file after all fixes for the issues found in the audit have been checked in:

- MD5 (2021-02-23.patch) = a0f5772a18c30d8c076c54b3a023ddb6

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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

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

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) [5], 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), 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 earning.farm protocol design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	2	■ ■
Informational	2	■ ■
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 1 medium-severity vulnerability, 2 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key earning.farm Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Unsafe ERC20 transfer() Calls	Business Logic	Fixed
PVE-002	Informational	Non-Governance-Based Admin of Time-Lock And Related Privileges	Security Features	Confirmed
PVE-003	Low	Unsafe Ownership Transition	Business Logic	Confirmed
PVE-004	Medium	Unsafe Minout Parameter in CRVExchange::handleExtraToken()	Business Logic	Fixed
PVE-005	Low	Incompatibility With Deflationary Tokens in CRVExchange::handleExtraToken()	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 Unsafe ERC20 transfer() Calls

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CRVExchange`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The earning.farm protocol pools together users' assets and deposits them into `Curve` to earn `CRV` token. The `earn_crv()` function in the `IUSDCPoolBase` contract will be called at least every 24 hours to harvest `CRV` and transfer them to the controller. This function will call `handleExtraToken()` to sell the tokens.

The token transfers are conducted via ERC20-compatible transferFrom() calls. However, while calling `IERC20(_token).transferFrom()`, the `CRVExchange` contract fails to check the return value as shown in line 61 below.

```
43  function handleExtraToken(address from, address target_token, uint256 amount) public{
44    uint256 maxOut = 0;
45    uint256 fdi = 0;
46    uint256 fpi = 0;
47
48    for(uint di = 0; di < dexs.length; di++){
49      for(uint pi = 0; pi < path_indexes.length; pi++){
50        if(path_from_addr(pi) != from   path_to_addr(pi) != target_token){
51          continue;
52        }
53        uint256 t = get_out_for_dex_path(di, pi, amount);
54        if( t > maxOut ){
55          fdi = di;
56          fpi = pi;
57          maxOut = t;
58        }
```

```
      }
    }
    IERC20(from).transferFrom(msg.sender, address(this), amount);
    IERC20(from).approve(dexs[fdi], amount);
    SushiUniInterface(dexs[fdi]).swapExactTokensForTokens(amount, 0, paths[path_indexes[
        fpi]], address(this), block.timestamp + 10800);

    uint256 target_amount = IERC20(target_token).balanceOf(address(this));
    IERC20(target_token).approve(address(msg.sender), target_amount);
    CFControllerInterface(msg.sender).refundTarget(target_amount);
  }
```

Listing 3.1: CRVExchange.sol

When the `_token` contract fails to revert for whatever reason, the caller of `transferFrom()` functions cannot ensure if the tokens are transferred successfully. In addition, certain ERC20 token contracts do not have a return value in its `transferFrom()` functions. To deal with these incompatibility issues, we suggest to use OpenZeppelin's `SafeERC20` library to accommodate various idiosyncrasies in current ERC20 implementations.

Recommendation Use OpenZeppelin's `SafeERC20` routines when interacting with ERC20 contracts.

Status This issue has been fixed by the team in the patch file: 87ca161.

3.2 Non-Governance-Based Admin of TimeLock And Related Privileges

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CFController`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In earning.farm protocol, , the owner of `CFController` contract plays a critical role in configuring parameters and withdrawing tokens. It can call the `pauseAndTransferTo()` funtion to withdraw assets from the `Curve` and sends it to the target. It can also set the important parameters `fee_pool`, `harvest_fee_ratio` by calling the `ChangeFeePool()`, `ChangeHarvestFee()`functions. These parameters are used to collect fees when users deposit tokens into vault.

```
  function pauseAndTransferTo(address _target) public onlyOwner{
    //pull out all target token
```

```
        uint256 cur = ICurvePool(current_pool).get_lp_token_balance();
        ICurvePool(current_pool).withdraw(cur);
        uint256 b = IERC20(target_token).balanceOf(address(this));

        IERC20(target_token).safeTransfer(_target, b);

        current_pool = address(0x0);
    }

    event ChangeFeePool(address old, address _new);
    function changeFeePool(address _fp) public onlyOwner{
        address old = fee_pool;
        fee_pool = _fp;
        emit ChangeFeePool(old, fee_pool);
    }

    event ChangeHarvestFee(uint256 old, uint256 _new);
    function changeHarvestFee(uint256 _fee) public onlyOwner{
        require(_fee < ratio_base, "invalid fee");
        uint256 old = harvest_fee_ratio;
        harvest_fee_ratio = _fee;
        emit ChangeHarvestFee(old, harvest_fee_ratio);
    }
```

Listing 3.2: Controller.sol

With great privilege comes great responsibility. In order to make the protocol safer, the owner of the contract should not be an EOA address but a `Timelock` or a `DAO` contract. So we recommend the team to transfer the ownership of the contract to a `Timelock`, `multi-signature` or a `DAO` contract after the deployment of the project.

Recommendation Transfer the ownership of the contract to a `Timelock`, `multi-signature` or a `DAO` contract.

Status This issue has been confirmed by the team. And they promised the owner will be a `multi-signature` contract.

3.3 Unsafe Ownership Transition

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Ownable`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In earning.farm, the `Ownable` contract is widely used for ownership management many contracts such as `CFETHController`, `CFController`, etc. When the contract owner needs to transfer the ownership to another address, she could invoke the `transferOwnership()` function with a `newOwner` address.

```
    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }

    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_contract_owner, newOwner);
        _contract_owner = newOwner;
    }
```

Listing 3.3: Ownable.sol

However, if the `newOwner` is not the exact address of the new owner (e.g., due to a typo), nobody could own that contract anymore.

Recommendation Implement a two-step ownership transfer mechanism that allows the new owner to claim the ownership by signing a transaction.

```
function transferOwnership(
    address newOwner
)
    external
    onlyOwner
{
    require(newOwner != address(0), "Owned: Address must not be null");
    require(candidateOwner != newOwner, "Owned: Same candidate owner");
    candidateOwner = newOwner;
}

function claimOwner()
    external
{
    require(candidateOwner == msg.sender, "Owned: Claim ownership failed");
    owner = candidateOwner;
    emit OwnerChanged(candidateOwner);
```

```
}
```

Listing 3.4: Ownable.sol

Status This issue has been confirmed by the team. However, this is not a security issue, the dev team decides to leave it as it is.

3.4 Unsafe Minout Parameter in CRVExchange::handleExtraToken()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: CRVExchange
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As we introduced in Section 3.1, the `IUSDCPoolBase::earn_crv()` function swaps the `CRV` earned to target token by calling `CRVExchange::handleExtraToken()`. The latter loops through the DEXs and finds out the one with best output. Then the function calls the DEX's `swapExactTokensForTokens()` to sell the earned `CRV`.

The `amountOutMin` parameter of `swapExactTokensForTokens()` is the minimum amount of output tokens that must be received for the transaction not to revert.

```
function handleExtraToken(address from, address target_token, uint256 amount) public{
  uint256 maxOut = 0;
  uint256 fdi = 0;
  uint256 fpi = 0;

  for(uint di = 0; di < dexs.length; di ++){
    for(uint pi = 0; pi < path_indexes.length; pi ++){
      if(path_from_addr(pi) != from   path_to_addr(pi) != target_token){
        continue;
      }
      uint256 t = get_out_for_dex_path(di, pi, amount);
      if( t > maxOut ){
        fdi = di;
        fpi = pi;
        maxOut = t;
      }
    }
  }
  IERC20(from).transferFrom(msg.sender, address(this), amount);
  IERC20(from).approve(dexs[fdi], amount);
```

```
    SushiUniInterface(dexs[fdi]).swapExactTokensForTokens(amount, 0, paths[path_indexes[
        fpi]], address(this), block.timestamp + 10800);

    uint256 target_amount = IERC20(target_token).balanceOf(address(this));
    IERC20(target_token).approve(address(msg.sender), target_amount);
    CFControllerInterface(msg.sender).refundTarget(target_amount);
  }
```

Listing 3.5: CRVExchange.sol

However, the `amountOutMin` here is set to 0. The front runners can buy the target token and sell the tokens after the admin have called the `IUSDCPoolBase::earn_crv()`. Since the `amountOutMin` is 0, the transaction harvesting the `CRV` won't revert if the output is small.

Recommendation Calculate the `amountOutMin` and submit it as a parameter.

Status This issue has been fixed by the team in the patch file: 87ca161.

3.5 Incompatibility With Deflationary Tokens in CRVExchange::handleExtraToken()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `CRVExchange`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As we introduced in Section 3.4, the `CRVExchange::handleExtraToken()` function sells `CRV` with specified DEXs and paths through `swapExactTokensForTokens()`. It swaps an exact amount of input tokens for as many output tokens as possible, along the route determined by the path.

```
  function handleExtraToken(address from, address target_token, uint256 amount) public{
    uint256 maxOut = 0;
    uint256 fdi = 0;
    uint256 fpi = 0;

    for(uint di = 0; di < dexs.length; di++){
      for(uint pi = 0; pi < path_indexes.length; pi++){
        if(path_from_addr(pi) != from  path_to_addr(pi) != target_token){
          continue;
        }
        uint256 t = get_out_for_dex_path(di, pi, amount);
        if( t > maxOut ){
          fdi = di;
```

```
          fpi = pi;
          maxOut = t;
        }
      }
    }
    IERC20(from).transferFrom(msg.sender, address(this), amount);
    IERC20(from).approve(dexs[fdi], amount);
    SushiUniInterface(dexs[fdi]).swapExactTokensForTokens(amount, 0, paths[path_indexes[
        fpi]], address(this), block.timestamp + 10800);

    uint256 target_amount = IERC20(target_token).balanceOf(address(this));
    IERC20(target_token).approve(address(msg.sender), target_amount);
    CFControllerInterface(msg.sender).refundTarget(target_amount);
  }
```

Listing 3.6: CRVExchange.sol

However, in the cases of deflationary tokens, the `swapExactTokensForTokens()` will fail. In order to supports tokens that take a fee on transfer, the uniswap V2 introduces `swapExactTokensForTokensSupportingFeeOnTransferTokens()`.

Recommendation Use `swapExactTokensForTokensSupportingFeeOnTransferTokens()` instead of `swapExactTokensForTokens()` .

Status This issue has been fixed by the team in the patch file: 87ca161.

4 Conclusion

In this audit, we have analyzed the earning.farm design and implementation. The system is a yield seeking protocol that pools together users' assets and deposits them into other Defi protocols to seek high yield. During the audit, we notice that the current code base is well structured and neatly organized, and those identified issues are promptly confirmed and fixed.

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

References

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

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

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

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

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

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

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