SMART CONTRACT AUDIT REPORT

for

MetaTrigger

Prepared By: Xiaomi Huang

PeckShield

June 07, 2022

Document Properties

Client	Meta Finance
Title	Smart Contract Audit Report
Target	MetaTrigger
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 07, 2022	Xuxian Jiang	Final Release
1.0-rc	June 02, 2022	Luck Hu	Release Candidate

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 smart contract source code of the MetaTrigger protocol, we outline in this 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 contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About MetaTrigger

MetaTrigger is a multi-chain yield aggregator powered by MetaFinance. It helps users to gain more cryptocurrency and increase their passive income in the DeFi area, including optimal staking pools matching strategies and auto-compounding strategies. And MetaTrigger is designed to provide users with a secure and decentralized approach to generate more earnings from their crypto assets more easily. Through a set of smart contracts and matching strategies, MetaTrigger automatically maximizes users' earnings from various pools and other mining opportunities within the DeFi ecosystem. This is more convenient and lower cost than doing it personally and creates more profitable opportunities.

Table 1.1: Basic Information of MetaTrigger

Item	Description
Name	Meta Finance
Website	https://metafinance.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 07, 2022

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

- https://github.com/MetaFinanceContract/trigger.git (99c6856)

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

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) [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 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 `MetaTrigger` 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	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 MetaTrigger Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack of Slippage Control in swapTokensForCake()	Time and State	Confirmed
PVE-002	Medium	Possible Assets Locked in SmartChef	Business Logic	Confirmed
PVE-003	Low	Proper Reward Tokens Accumulation For Swap	Coding Practices	Confirmed
PVE-004	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 Lack of Slippage Control in swapTokensForCake()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MetaFinanceTriggerPool`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `MetaFinanceTriggerPool` contract accepts users deposit of `CAKE` and re-invests them to an array of `smartChef` pools to earn rewards. After the rewards are withdrawn from the `smartChef` pools, they will be swapped back to `CAKE`.

To elaborate, we show below the code snippet of the `swapTokensForCake()` routine. As the name indicates, it's designed to swap the input token to `CAKE`. The swap is completed by `PancakeRouter` which supports an input of the expected minimum amount of the received `CAKE` token.

```
    function swapTokensForCake(
        IERC20Metadata tokenAddress,
        address[] memory path,
        uint256 oldBalanceOf
    ) private {
        uint256 tokenAmount = tokenAddress.balanceOf(address(this)).sub(oldBalanceOf);

        if (tokenAmount < proportion) return;
        tokenAddress.safeApprove(address(pancakeRouterAddress), 0);
        tokenAddress.safeApprove(address(pancakeRouterAddress), tokenAmount);

        // address(this) Reward token -> address(uniswapV2Pair) wbnb
        // address(uniswapV2Pair) wbnb -> address(uniswapV2Pair) cake
        // address(uniswapV2Pair) cake -> address(this)
        pancakeRouterAddress.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            tokenAmount,
            1, // accept any amount of cake
            path,
```

```
244                 address(this),
245                 block.timestamp + 60
246             );
247     }
```

Listing 3.1: MetaFinanceTriggerPool::swapTokensForCake()

However, it comes to our attention that there is no slippage control in place (line 242), which opens up the possibility for front-running and potentially results in a smaller converted amount. Note that 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. 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 `Pancake`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich arbitrage to better protect the interests of users.

Status This issue has been confirmed by the team. And the team can accept current minimum amount in order to save certain amount of transaction fees and for business needs.

3.2 Possible Assets Locked in SmartChef

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MetaFinanceTriggerPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `MetaFinanceTriggerPool` contract provides an interface (i.e. `projectPartyEmergencyWithdraw()`) for project administrator to withdraw user assets from the `SmartChef` pool at emergency. Our analysis shows its current implementation needs to be improved.

To elaborate, we show below the code snippets from the `MetaFinanceTriggerPool` contract. The `projectPartyEmergencyWithdraw()` routine is used to withdraw user assets from the given `smartChef_` and update the protocol lock with the given `urgent_`. After the withdraw, it reduces the `totalPledgeValue` which is the total pledge that is deposited in all supported `smartChef` pools.

```
    function projectPartyEmergencyWithdraw(ISmartChefInitializable smartChef_, bool
        urgent_) external nonReentrant onlyRole(PROJECT_ADMINISTRATOR) {
        if (totalPledgeAmount != 0) {
            smartChef_.emergencyWithdraw();
            totalPledgeValue = totalPledgeValue.sub(storageQuantity[smartChef_]);
            storageQuantity[smartChef_] = 0;
            if (urgent_)
                urgent = urgent_;
        }
    }
```

Listing 3.2: MetaFinanceTriggerPool::projectPartyEmergencyWithdraw()

It comes to our attention that, the `totalPledgeValue` may be reduced to be smaller than the `proportion` value. Based on this, the `updateMiningPool()` will not trigger withdraw from all `smartChef` pools. And if the `totalPledgeValue` value becomes larger than the `proportion` value again in the `reinvest()` routine, it will overwrite the `storageQuantity[i]` which records the amount of deposit in each `smartChef` pool. As a result of this, the original deposit amounts are lost, and the deposited assets have no way to be withdrawn anymore.

```
    /**
    * @dev Update mining pool
    * @notice Batch withdraw,
    *          and will experience token swap to cake token,
    *          and increase the rewards for all users
    */
    function updateMiningPool() private nonReentrant {
        cakeTokenBalanceOf = cakeTokenAddress.balanceOf(address(this));
        if (totalPledgeValue > proportion && smartChefArray.length > 0) {
            uint256 length = smartChefArray.length;
            address[] memory path = new address[](3);
            path[1] = address(wbnbTokenAddress);
            path[2] = address(cakeTokenAddress);
            for (uint256 i = 0; i < length; ++i) {
                uint256 rewardTokenBalanceOf = IERC20Metadata(smartChefArray[i].
                    rewardToken()).balanceOf(address(this));
                if (storageQuantity[smartChefArray[i]] != 0) {
                    smartChefArray[i].withdraw(storageQuantity[smartChefArray[i]]);
                    path[0] = smartChefArray[i].rewardToken();
                    swapTokensForCake(IERC20Metadata(path[0]), path,
                        rewardTokenBalanceOf);
                }
            }

            uint256 haveAward = ((cakeTokenAddress.balanceOf(address(this))).sub(
                totalPledgeValue)).sub(cakeTokenBalanceOf);

            if (totalPledgeAmount != 0) {
                (uint256 userRewards, uint256 exchequerRewards) = totalUserRewards(
                    haveAward);
                exchequerAmount = exchequerAmount.add(exchequerRewards);
```

```
                takenTransfer(address(this), address(this), userRewards);
            } else {
                exchequerAmount = exchequerAmount.add(haveAward);
            }
        }
    }

    /**
    * @dev Bulk pledge
    */
    function reinvest() private nonReentrant {
        totalPledgeValue = (cakeTokenAddress.balanceOf(address(this))).sub(
            cakeTokenBalanceOf);
        if (totalPledgeValue > proportion && smartChefArray.length > 0) {
            uint256 _frontProportionAmount = 0;
            uint256 _arrayUpperLimit = smartChefArray.length;
            for (uint256 i = 0; i < _arrayUpperLimit; ++i) {
                if (i != _arrayUpperLimit - 1) {
                    storageQuantity[smartChefArray[i]] = (totalPledgeValue.mul(
                        storageProportion[smartChefArray[i]])).div(proportion);
                    _frontProportionAmount += storageQuantity[smartChefArray[i]];
                }
                if (i == _arrayUpperLimit - 1)
                    storageQuantity[smartChefArray[i]] = totalPledgeValue.sub(
                        _frontProportionAmount);
            }
            for (uint256 i = 0; i < _arrayUpperLimit; ++i) {
                cakeTokenAddress.safeApprove(address(smartChefArray[i]), 0);
                cakeTokenAddress.safeApprove(address(smartChefArray[i]), storageQuantity
                    [smartChefArray[i]]);
                smartChefArray[i].deposit(storageQuantity[smartChefArray[i]]);
            }
        }
    }
```

Listing 3.3: MetaFinanceTriggerPool.sol

What's more, the same issue exists in the `setProportion()` routine where the `proportion` is updated which could also impact the deposit/withdraw with the `smartChef` pools.

Recommendation Revisit the above mentioned routines to properly maintain the deposit amounts in all `smartChef` pools.

Status This issue has been confirmed by the team. And the team clarifies that when the emergency withdraw is triggered, all user-executable operations will be permanently suspended, and the contract will also be discarded.

3.3 Proper Reward Tokens Accumulation For Swap

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `MetaFinanceTriggerPool`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1099 [1]

Description

As mentioned in Section 3.1, the `MetaFinanceTriggerPool` contract invests users deposit of `CAKE` into an array of `smartChef` pools to earn rewards. After the rewards are withdrawn from these `smartChef` pools, they will be swapped back to `CAKE` via `pancakeRouter`.

To elaborate, we show below the code snippet of the `swapTokensForCake()` routine. As the name indicates, it's designed to swap the input token (given by `tokenAddress`) to `CAKE`. This routine takes a parameter `oldBalanceOf` which is the balance of the given token before the withdrawal from the `smartChef` pool. And the token amount to be swapped is calculated via `tokenAmount = tokenAddress.balanceOf(address(this)).sub(oldBalanceOf)`. As a result, the `tokenAmount` is the new received token amount from the latest withdraw. However, it comes to our attention that, there's a validation check (line 233) which will directly return if the `tokenAmount` is below the `proportion`. In this case, the new received reward tokens will not be swapped to the target `CAKE` anymore. Based on this, it's suggested to accumulate the amount of each reward token until it's large enough to be swapped.

```
226   function swapTokensForCake(
227       IERC20Metadata tokenAddress,
228       address[] memory path,
229       uint256 oldBalanceOf
230   ) private {
231       uint256 tokenAmount = tokenAddress.balanceOf(address(this)).sub(oldBalanceOf);
232
233       if (tokenAmount < proportion) return;
234       tokenAddress.safeApprove(address(pancakeRouterAddress), 0);
235       tokenAddress.safeApprove(address(pancakeRouterAddress), tokenAmount);
236
237       // address(this) Reward token -> address(uniswapV2Pair) wbnb
238       // address(uniswapV2Pair) wbnb -> address(uniswapV2Pair) cake
239       // address(uniswapV2Pair) cake -> address(this)
240       pancakeRouterAddress.swapExactTokensForTokensSupportingFeeOnTransferTokens(
241           tokenAmount,
242           1, // accept any amount of cake
243           path,
244           address(this),
245           block.timestamp + 60
246       );
```

```
}
```

Listing 3.4: MetaFinanceTriggerPool::swapTokensForCake()

Recommendation Revisit the above mentioned routine to properly accumulate the amount of each reward token until it's large enough to be swapped.

Status This issue has been confirmed by the team. And the team has considered this case at the beginning of the project.

3.4 Trust Issue of Admin Keys

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

Description

In the `MetaTrigger`, there are certain privileged administrators that play critical roles in governing and regulating the system-wide operations (e.g., set the fee rate and withdraw fees). In the following, we examine the privileged accounts and the related privileged accesses in current contracts.

```
/**
* @dev Modify the fee ratio
* @param newTreasuryRatio_ New treasury fee ratio
*/
function setFeeRatio(uint256 newTreasuryRatio_) external beforeStaking nonReentrant
    onlyRole(DATA_ADMINISTRATOR) {
    require(newTreasuryRatio_ <= proportion, "MFTP:E8");
    if (newTreasuryRatio_ != 0) treasuryRatio = newTreasuryRatio_;
}

/**
* @dev claim Tokens to treasury
*/
function claimTokenToTreasury() external beforeStaking nonReentrant onlyRole(
    MONEY_ADMINISTRATOR) {
    cakeTokenAddress.safeTransfer(metaFinanceClubInfo.treasuryAddress(),
        exchequerAmount);
    exchequerAmount = 0;
}
```

Listing 3.5: Example Privileged Operations in `MetaFinanceTriggerPool.sol`

```
/**
* @dev Set treasury address
* @param newTreasury_ New treasury address
*/
function setTreasuryAddress(address newTreasury_) external onlyRole(DATA_ADMINISTRATOR)
    {
    treasuryAddress = newTreasury_;
}
```

Listing 3.6: Example Privileged Operations in `MetaFinanceClubInfo.sol`

There are still other privileged routines not listed here. We point out that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol 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 Making the above privileges explicit among protocol users.

Status This issue has been confirmed by the team. And the team clarifies that: The permissions granted in this project have been strictly reviewed, and some special permissions will be handed over to a multi-sig account. After all permissions are given, the project will give up the highest administrator permissions.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `MetaTrigger` protocol, which is a multi-chain yield aggregator powered by `MetaFinance`. It helps users to gain more cryptocurrency and increase their passive income in the `DeFi` area, including optimal staking pools matching strategies and auto-compounding strategies. The current code base is well organized and those identified issues are promptly confirmed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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 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.