SMART CONTRACT AUDIT REPORT

for

Furucombo Wallets & FuruGelato

Prepared By: Yiqun Chen

PeckShield

September 30, 2021

PeckShield

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 30, 2021	Xuxian Jiang	Final Release
1.0-rc	September 5, 2021	Xuxian Jiang	Release Candidate

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

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Furucombo wallet system and the FuruGelato 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 several issues. This document outlines our audit results.

1.1 About Furucombo Wallets/FuruGelato

Furucombo is a tool developed for end-users to optimize their DeFi strategy with simple, convenient, and visualized drag-and-drop operations. The audited Furucombo wallet system helps users easily manage their assets. Users can interact with multiple contracts in a single transaction, manage wallet authority, and execute automation tasks. And the FuruGelato protocol allows for automate task executions through the user's DSProxy-based wallets. Specifically, the user's task can be executed once the pre-defined condition is satisfied.

Table 1.1: Basic Information of Furucombo Wallets/FuruGelato

Item	Description
Name	Furucombo
Website	https://furucombo.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 30, 2021

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

- https://github.com/dinngodev/furucombo-smart-wallet.git (8852383)
- https://github.com/dinngodev/furucombo-gelato.git (1270190)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/dinngodev/furucombo-smart-wallet.git (3368d78)
- https://github.com/dinngodev/furucombo-gelato.git (00a8919)

1.2 About PeckShield

PeckShield Inc. [7] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 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) 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 1.3: 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

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 `Furucombo` wallet system and the `FuruGelato` 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	0	
Low	2	■ ■
Undetermined	1	■
Total	3	

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 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Furucombo Wallets/FuruGelato Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Suggested Whitelisting of Execution Targets in TaskExecutor	Security Features	Resolved
PVE-002	Low	Payment Fee Issue in FuruGelato::exec()	Business Logic	Resolved
PVE-003	Low	Trust Issue Of Admin Keys	Security Features	Resolved

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.

PeckShield

3 | Detailed Results

3.1 Suggested Whitelisting of Execution Targets in TaskExecutor

- ID: PVE-001
- Severity: Undetermined
- Likelihood: Low
- Impact: Low
- Target: TaskExecutor
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

The Furucombo wallet system is designed to help users to better manage their assets in terms of seamlessly interacting with multiple contracts in a single transaction, conveniently managing wallet authority, and reliably executing automation tasks. Specifically, the TaskExecutor contract extends the widely-used DSProxy with advanced features to support the interaction with complex DeFi protocols into so-called cubes. Users may setup inputs and outputs as well as the order of the cubes, then bundle all the cubes into one transaction and sends out.

```
    function _execs(
        address[] memory tos,
        bytes32[] memory configs,
        bytes[] memory datas
    ) internal {
        bytes32[256] memory localStack;
        uint256 index = 0;

        require(
            tos.length == datas.length,
            "TaskExecutor: Tos and datas length inconsistent"
        );
        require(
            tos.length == configs.length,
            "TaskExecutor: Tos and configs length inconsistent"
        );
```

```
        for (uint256 i = 0; i < tos.length; i++) {
            bytes32 config = configs[i];

            if (config.isDelegateCall()) {
                // Delegate call case

                // Trim params from local stack depend on config
                _trimParams(datas[i], config, localStack, index);

                // Execute action by delegate call
                bytes memory result =
                    tos[i].functionDelegateCall(
                        datas[i],
                        "TaskExecutor: low-level delegate call failed"
                    );

                // Store return data from action to local stack
                index = _parseReturn(result, config, localStack, index);
            } else {
                // Function Call case

                // Decode eth value from data
                (uint256 ethValue, bytes memory _data) =
                    _decodeEthValue(datas[i]);

                // Trim params from local stack depend on config
                _trimParams(_data, config, localStack, index);

                // Execute action by call
                bytes memory result =
                    tos[i].functionCallWithValue(
                        _data,
                        ethValue,
                        "TaskExecutor: low-level call with value failed"
                    );

                // Store return data from action to local stack depend on config
                index = _parseReturn(result, config, localStack, index);
            }
        }
    }
```

Listing 3.1: TaskExecutor::_execs()

We point out that each action for execution involves the `call` or `delegatecall` with the intended target `tos[i]`. With that, as a security precaution, we suggest to whitelist these targets before they can be trusted for interaction.

Recommendation Add a whitelist feature to ensure only trusted targets can be allowed for interaction.

Status This issue has been confirmed. The team clarifies that the `TaskExecutor` component is more like a tool to extend user `DSProxy` functionality. Even users don't use `TaskExecutor` to delegate call other actions. They could also use `execute()` of `DSProxy` directly to delegate call other actions. In this case, we still could not prevent this situation when we add the `Whitelist` in `TaskExecutor`.

3.2 Payment Fee Issue in FuruGelato::exec()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FuruGelato`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `FuruGelato` protocol, a specific task is referred as an actual execution being triggered through a `DSProxy`. The task can be verified through a resolver when it is created or executed. In the examination of the `exec()` logic, we notice the given fee needs to be pre-agreed and trusted among involved parties.

To elaborate, we show below the related `exec()` function that is designed to execute the task created by `_proxy` through the given `_resolverAddress`. This function has an affiliated modifier `gelatofy`, which not only enforces the caller to be `gelato`, but also collects the required `fee` back to `gelato`.

```
    function exec(
        uint256 _fee,
        address _proxy,
        address _resolverAddress,
        bytes calldata _executionData
    )
        external
        override
        gelatofy(_fee, ETH)
        onlyValidResolver(_resolverAddress)
        onlyValidDSProxy(_proxy)
    {
        bytes32 taskId = getTaskId(_proxy, _resolverAddress, _executionData);
        require(isValidTask(taskId), "FuruGelato: exec: invalid task");
        // Fetch the action to be used in dsproxy's ‘execute()‘.
        address action = Resolver(_resolverAddress).action();

        require(
            _proxy == taskCreator[taskId],
            "FuruGelato: exec: No task found"
        );

```

```
        try IDSProxy(_proxy).execute(action, _executionData) {} catch {
            revert("FuruGelato: exec: execute failed");
        }

        require(
            Resolver(_resolverAddress).onExec(_proxy, _executionData),
            "FuruGelato: exec: onExec() failed"
        );

        emit ExecSuccess(_fee, ETH, _proxy, taskId);
    }
```

Listing 3.2: `FuruGelato::exec()`

```
    modifier gelatofy(uint256 _amount, address _paymentToken) {
        require(msg.sender == gelato, "Gelatofied: Only gelato");
        _;
        if (_paymentToken == ETH) {
            (bool success, ) = gelato.call{value: _amount}("");
            require(success, "Gelatofied: Gelato fee failed");
        } else {
            SafeERC20.safeTransfer(IERC20(_paymentToken), gelato, _amount);
        }
    }
```

Listing 3.3: `Gelatofied::gelatofy()`

It comes to our attention that the collected `Gelato fee` is directly specified in the input argument without any sanity checks. In other words, this calling `gelato` needs to be fully trusted.

Recommendation Restrict the collected fee within a reasonable range instead of allowing for arbitrate charge.

Status This issue has been confirmed with the team.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the audited systems, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., whitelist/blacklist setup and parameter configuration).

In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    /// @notice Ban the dsProxy from being able to be executed.
    /// @param _dsProxy The dsProxy to be banned.
    function banDSProxy(address _dsProxy) external onlyOwner {
        _blacklistedDSProxies[_dsProxy] = true;

        emit DSProxyBlacklistAdded(_dsProxy);
    }

    /// @notice Unban the dsProxy.
    /// @param _dsProxy The dsProxy to be unbanned.
    function unbanDSProxy(address _dsProxy) external onlyOwner {
        require(!isValidDSProxy(_dsProxy), "Not banned");
        _blacklistedDSProxies[_dsProxy] = false;

        emit DSProxyBlacklistRemoved(_dsProxy);
    }
```

Listing 3.4: BunnyParkCakeBPPoolV2::banDSProxy()/unbanDSProxy()

Notice 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 onlyOwner privileges explicit or raising necessary awareness among protocol users.

Recommendation Making these onlyOwner privileges explicit among protocol users.

Status This issue has been confirmed with the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Furucombo` wallet system and the `FuruGelato` protocol. They are designed to help users to better manage their assets in terms of seamlessly interacting with multiple contracts in a single transaction, conveniently managing wallet authority, and reliably executing automation tasks. The current code base is well organized and 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.

PeckShield

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.

PeckShield