SMART CONTRACT AUDIT REPORT

for

FEG SmartDeFi

Prepared By: Xiaomi Huang

PeckShield

May 15, 2024

Document Properties

Client	FEG
Title	Smart Contract Audit Report
Target	FEG SmartDeFi
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 15, 2024	Xuxian Jiang	Final Release
1.0-rc	May 12, 2024	Xuxian Jiang	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 FEG 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 FEG

The FEG ecosystem is poised to transform the DeFi landscape, empowering users to seamlessly manage their finances, launch innovative projects, trade with unparalleled security, and benefit from the growth and success of the entire ecosystem. The FEG has its asset-backed & passive income earning governance token, operating on both Ethereum and Binance Smart Chain. The FEG asset-backing creates a store-of-value with an ever-rising baseline. FEG LGEs are token-less by allowing users invest native coins, which are then matched with tokens as vested LP on a vesting schedule. The investor can later claim to remove LP, by burning the SD token for backing and then giving native coins as well as backing to the investor. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The FEG Protocol

Item	Description
Name	FEG
Website	https://fegtoken.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 15, 2024

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

- https://github.com/FEGrox/SmartDeFi_Final.git (bcc75d3)

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

- https://github.com/FEGrox/SmartDeFi_Final.git (f5598b2)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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.

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

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) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the FEG 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	
Undetermined	1	
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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key FEG SmartDeFi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited giveKarma() Logic in SD_Deployer	Business Logic	Resolved
PVE-002	Undetermined	Possible Tick Manipulation Against Anti-Frontrunning Protection	Business Logic	Mitigated
PVE-003	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Revisited giveKarma() Logic in SD_Deployer

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SD_Deployer`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `FEG` protocol has a core `SD_Deployer` contract that allows for the creation of a new `SD` token. This `SD_Deployer` contract also supports users to submit suggestions, comments, or donate native tokens. In the process of reviewing the donation logic, we notice the related routine can be improved.

To elaborate, we show below the implementation of the related `giveKarma()` routine. It has two places for improvement. The first place is the `break` statement (line 450) inside the `for`-loop. The `break` statement is guarded with the `if(d)` condition (line 449), which is always evaluated to be true and hence can be removed.

The second place is part of the `karma[sd]` assignment statement (line 455), which should be improved as `karma[sd] = choice == 0 ? karma[sd] + 1 : karma[sd] > 0 ? karma[sd] - 1 : 0;`.

```
371     function giveKarma(address sd, uint256 choice) external payable nonReentrant() {
372         require(msg.value == karmaDonation[sd], "min");
373         require(block.timestamp > lastKarma[msg.sender][sd] + 24 hours, "1 day");
374         bool d;
375         (uint256[] memory _balances, uint256[] memory blockNumbers) = Reader(sd).
                allLastBalance(msg.sender);
376         for(uint256 i = 0; i < _balances.length; i++) {
377             if(blockNumbers[i] < block.number - 5 && IERC20(sd).balanceOf(msg.sender) >=
                     _balances[i]) {
378                 d = true;
379                 if(d) {
380                     break;
381                 }
```

```
            }
        }
        require(d, "!5blocks");
        karma[sd] = choice == 0 ? karma[sd] += 1 : karma[sd] > 0 ? karma[sd] -= 1 : 0;
        if(karmaDonation[sd] > 0) {
            SafeTransfer.safeTransferETH(donationLocation[sd], msg.value);
        }
        lastKarma[msg.sender][sd] = block.timestamp;
        emit GiveKarma(sd, choice);
    }
```

Listing 3.1: `SD_Deployer::giveKarma()`

Recommendation Improve the above routine by optimizing its logic for simplified implementation. Note the `replyTicket()` shares the same issue.

Status The issue has been resolved by the following commit: `f5598b2`.

3.2 Possible Tick Manipulation Against Anti-Frontrunning Protection

- ID: PVE-002
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `SD_Deployer`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The new `SD` token has a built-in anti-frontrunning protection mechanism that keeps track of the `WETH/FEG` balances in respective liquidity pairs. In the process of examining its effectiveness, we notice a possible issue that may compromise the intended anti-frontrunning protection.

In particular, we show below the related implementation of the `SD_Deployer::setTick()` routine. It has a rather straightforward logic in tracking the `WETH` balance in the backing pair of `SD` token as well as the `FEG` balance in the `FEG-WETH` pair. We notice the reserves are saved if the tracked balances are different from the last saving. With that, it is possible to simply sandwich the `setTick()` call with the huge swap between `backingAsset` and `WETH` to the backing pair. Each time, we can manipulate the swap amount to write the intended reserve amount into the internal `tick` array (line 777), reducing the effectiveness of the built-in anti-frontrunning mechanism.

```
763    function setTick(address who) external {
764        address DR = dataread;
765        if(!Reader(DR).tickOn(who)) {
```

```
            address ba = Logic(who).backingAsset();
            address weth = wETH();
            require(Reader(DR).isProtocol(who) && isSD(who), "caller");
            address fac = IUniswapV2Router01(Reader(who).UNISWAP_V2_ROUTER()).factory();
            address t = ba == weth ? Reader(DR).uniswapV2Pair(who) : IUniswapV2Router01(
                fac).getPair(wETH(), ba);
            if(IERC20(ba).balanceOf(t) > 0) {
                uint256 k1 = tick[t].length > 0 ? tick[t][tick[t].length - 1] : 0;
                (uint112 reserve0, uint112 reserve1,) = IPair(t).getReserves();
                uint256 reserve = ba == IPair(t).token0() ? reserve0 : reserve1;
                if(reserve != k1) {
                    tick[t].push(reserve);
                }
            }
        }
    }
```

Listing 3.2: `SD_Deployer::setTick()`

Recommendation Revisit the above anti-frontrunning logic to ensure it may not be bypassed.

Status The issue has been mitigated. The team removes the need of tracking `FEG`'s ticks to further reduce the associated risk. In addition, there is no user that can call this function and the user would be subject to any taxes on the transaction making it hard to really benefit from manipulating.

3.3 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SD_Deployer`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `SD_Deployer` as an example, the `confirmKYC()` function (see the code snippet below) is provided to externally call a contract. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 586) start before effecting the update on internal state (lines 588 and 590), hence violating the principle. (Fortunately, this function has been properly guarded with the `nonReentrant` modifier.)

```
    function confirmKYC(address sd, address user) external {
        require(isSupport[sd][msg.sender], "not support");
        require(!kyc[sd][user].confirmed && kyc[sd][msg.sender].time > 0, "already");
        KYCopen[sd] -= 1;
        uint256 don = kyc[sd][user].donation;
        if(don > 0) {
            SafeTransfer.safeTransferETH(msg.sender, don);
            heldDonation[sd] -= don; // underflow desired as require >= 0
            kyc[sd][user].donation = 0;
        }
        kyc[sd][user].confirmed = true;
        emit ConfirmKYC(sd, user);
    }
```

Listing 3.3: `SD_Deployer::confirmKYC()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle. Note the use of `nonReentrant` modifier is already in place to block the reentrancy risk.

Status The issue has been resolved by the following commit: `f5598b2`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

The `FEG` token instantiation and management is designed with a privileged account, i.e., `owner`, that play a critical role in governing and regulating the system-wide operations (e.g., configure parameters, adjust various fees, set maturity delay, and perform emergency withdraw). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setWhitelist(bool _bool) external {
        require(msg.sender == owner); ...
    }

    // function for adding bulk initial whitelist data
    function addBulkWhitelist(address[] memory _list) external {
        require(msg.sender == owner); ...
    }

    // function to edit whitelist spot
    function editWhitelist(address user, bool _live) external {
        require(msg.sender == owner && user != owner); ...
    }
    function saveTokens(address token) public nonReentrant {
        require(msg.sender == owner); ...
    }
```

Listing 3.4: Example Privileged Operations in `LGE_Logic`

We understand the need of the privileged function for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is plain `EOA` account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed `DAO`.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated with the use of a multi-sig for the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the FEG protocol, which is poised to transform the DeFi landscape by empowering users to seamlessly manage their finances, launch innovative projects, trade with unparalleled security, and benefit from the growth and success of the entire ecosystem. The FEG has its asset-backed & passive income earning governance token, operating on both Ethereum and Binance Smart Chain. The FEG asset-backing creates a store-of-value with an ever-rising baseline. FEG LGEs are token-less by allowing users invest native coins, which are then matched with tokens as vested LP on a vesting schedule. The investor can later claim to remove LP, by burning the SD token for backing and then giving native coins as well as backing to the investor. During the audit, we notice that the current code base is well organized and those identified issues are promptly mitigated and fixed.

Meanwhile, 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.