SMART CONTRACT AUDIT REPORT

for

FEG Staking

Prepared By: Xiaomi Huang

PeckShield

July 10, 2023

Document Properties

Client	FEG
Title	Smart Contract Audit Report
Target	FEG Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 10, 2023	Xuxian Jiang	Final Release
1.0-rc	June 28, 2023	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 Staking support, 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 Staking

FEG Token is the asset-backed & passive income earning governance token of its fully decentralized ecosystem, operating on both Ethereum and Binance Smart Chain. The FEG asset-backing creates a store-of-value with an ever-rising baseline. FEG stakers can earn passive income as all ecosystem-generated fees are used for FEG staking rewards. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The Migrator Protocol

Item	Description
Name	FEG
Website	https://fegtoken.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 10, 2023

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

- https://github.com/FEGrox/SD-StakeDeployer.git (94eeae6)

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

- https://github.com/FEGrox/SD-StakeDeployer.git (4734d6b)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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

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

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) [10], 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 Staking` 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	3	■■■
Low	4	■■■■
Informational	0	
Total	7	

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 3 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key FEG Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Staking Logic in StakingInterface::stake()	Business Logic	Resolved
PVE-002	Medium	Improper Withdrawal Logic In StakingLogic	Business Logic	Resolved
PVE-003	Low	Improved Sanity Checks Upon Parameter Changes	Coding Practices	Resolved
PVE-004	Low	Removal of Redundant Data And Code	Coding Practices	Resolved
PVE-005	Low	Improved Precision By Multiplication And Division Reordering	Coding Practices	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Bypassed Token Validation For Arbitrary Reward Addition	Security Features	Confirmed

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 Staking Logic in StakingInterface::stake()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakingInterface`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Staking` support allows users earn `passive income` as the `FEG` ecosystem-generated fees feed the staking rewards. While examining the current staking logic, we notice the implementation may need to be revisited.

In particular, we show below the related implementation of the `StakingInterface::stake()` routine. It has a rather straightforward logic in transfering the staking token from the user into itself, which then immediately forwards the staking token to the `stakeLogic` contract. It comes to our attention that the staking token is a reflection one which might have a transfer fee. As a result, the forwarding amount may not be the same as the initial staking amount. In other words, we need to properly use the actual received amount for the forwarding, instead of the initial amount. Note the `StakingLogic` contract also has a `stake()` routine, which shares the same issue.

Moreover, we notice the staking token is eventually saved in the `StakingInterface` contract. As a result, the staking funds via the following `StakingInterface::stake()` routine will in essence tranfer the reflection token back and forth between `StakingInterface` and `StakingLogic`, which should be avoided to save unnecessary transfer fee.

```
    function stake(uint256 amount) public nonReentrant returns(uint256 poolAmountOut){
        SafeTransfer.safeTransferFrom(IERC20(SD), msg.sender, address(this), amount);
        SafeTransfer.safeTransfer(IERC20(SD), stakeLogic, amount);
        poolAmountOut = StakeLogics(stakeLogic).stake(msg.sender, amount);
        emit STAKED(msg.sender, amount);
        emit Transfer(msg.sender, address(this), poolAmountOut);
```

```
        return poolAmountOut;
    }
```

Listing 3.1: `StakingInterface::stake()`

Recommendation Revisit the above staking logic to ensure the staking amount is properly accounted for and no extra transfer fee is charged.

Status The issue has been resolved by setting the related stake interface and stake logic contacts as feeless.

3.2 Improper Withdrawal Logic In StakingLogic

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakingLogic`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The staking support allows the users to deposit the intended tokens into the staking contract. In the meantime, it also allows the user to redeem the staked funds. While examining the current unstaking logic, we notice the related logic also needs to be revisited.

To elaborate, we show below the implementation of the related `withdraw()` routine. The current unstaking logic computes the withdraw fee and the sacrifice amount, next calculates the actual amount for withdrawal, i.e., `tokensToWithdraw = tokenAmountOut - (wdFee - sac)` (line 1312). Apparently, the current approach has an implicit assumption that `wdFee>=sac`, which unfortunately may not be the case. In the corner case where the sacrifice amount is larger than the withdraw fee, the current unstaking execution may be unexpectedly reverted!

In addition, if the protocol parameter `burnWDFee` indicates the need of not burning the withdraw fee, the related fee is supposed to sent to the pool. However, it is not consistent with the current implementation, which simply keeps the fee in the withdrawing user account (lines 1315-1318).

```
1268    function withdraw(address user, uint256 amt) public pse returns (uint256
            tokenAmountOut){
1269        if(msg.sender != parent){
1270            require(msg.sender == user, "Not user");
1271        }
1272        if(matureDelay){
1273            require(block.timestamp >= stakers[user].stakeTime + delay, "Not mature");
1274        }
1275        if(DR(SD2(SD).DATA_READ()).feeConverter() != address(0)){
```

```
            protCont(DR(SD2(SD).DATA_READ()).feeConverter()).cont(SD, 0);
        }
        if(userRewardCheck(user)){
            claimAllReward(user);
            require(!userRewardCheck(user),"cl");
        }
        uint256 totalSD = IERC20(SD).balanceOf(parent);
        uint256 tokens = calcPoolInGivenSingleOut(
                            totalSD,
                            bmul(BASE, 25),
                            _totalSupply,
                            bmul(BASE, 25),
                            amt,
                            0
                        );
        require(tokens <= balanceOf(user), "You don't have enough");
        tokenAmountOut = calcSingleOutGivenPoolIn(
                            totalSD,
                            bmul(BASE, 25),
                            _totalSupply,
                            bmul(BASE, 25),
                            tokens,
                            0
                        );
        uint256 wdFee = tokenAmountOut.div(1000).mul(withdrawFee);
        if(burnWDFee){
        stakeInterface(parent).sendTokens(dead, wdFee);
        }

        uint256 sac;
        if(sacrificeEnabled){
            sac = tokenAmountOut.div(1000).mul(stakers[user].sacrificeLevel);
            stakeInterface(parent).sendTokens(dead, sac);
            tokenAmountOut -= sac;
        emit USERSACRIFICE(user, sac);
        }
        uint256 tokensToWithdraw = tokenAmountOut - (wdFee - sac);
        _pullPoolShare(user, tokens);
        _burn(tokens);
        if(!burnWDFee && wdFee != 0){
            amt -= wdFee;
        }
        stakers[user].amtStaked -= amt;
        totalStakedSD -= amt;
        if(balanceOf(user) == 0) {
            stakers[user].amtStaked = 0;
            stakers[user].stakeTime = 0;
            stakers[user].initialized = false;
        }
        stakeInterface(parent).sendTokens(user, tokensToWithdraw);
        emit UNSTAKED(user, tokensToWithdraw);
        return tokensToWithdraw;
```

```
}
```

Listing 3.2: StakingLogic :: withdraw()

Recommendation Revise the above unstaking logic to avoid arithmetic underflow and return the fee to the current pool (if the protocol is configured to do so).

Status The issue has been fixed by the following commit: 4734d6b.

3.3 Improved Sanity Checks For System Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The FEG Staking support is no exception. Specifically, if we examine the stakeLogic contract, it has defined a number of protocol-wide risk parameters, such as depositFee and withdrawFee. In the following, we show the corresponding routines that allow for their changes.

```
function setDepositFee(uint256 amt) external {
    if(msg.sender != parent){require(msg.sender == owner);}
    depositFee = amt;
}

 // ---------------------------------------------------------------------
 // Allows setting withdraw fee
 // ---------------------------------------------------------------------
function setWithdrawalFee(uint256 amt) external {
    if(msg.sender != parent){require(msg.sender == owner);}
    withdrawFee = amt;
}
```

Listing 3.3: stakeLogic::setDepositFee()/setWithdrawalFee()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of depositFee may charge unreasonably high fee in the staking operation.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been addressed by implementing 20% max fee for deposit and withdraw fee implementing 60% max sacrifice which will allow for no underflow – as shown in the following commit: `b4b05e7`.

3.4 Removal of Redundant Data And Code

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

The `FEG Staking` contracts make good use of a number of reference contracts, such as `ERC20`, `SafeTransfer`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `StakingLogic` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `StakingLogic` contract, there is a `SafeMath` library to support safe `math` operations by preventing common overflow or underflow issues. However, it is no longer needed if the current compiler is `Solidity version 0.8.0` and onward since the same support has been integrated by default in the compiler.

```
    struct REWARDS{
        bool live;
        uint256 round;
        uint256 totalDividends;
        uint256 totalClaimedReward;
        uint256 totalUnclaimedReward;
        uint256 totalRewards;
        uint256 syncLevel;
    }

    struct REWARDROUNDS{
        uint256 payouts;
        uint256 payoutTS;
        uint256 dpt;
    }
```

Listing 3.4: Key Data Structure Used in `StakingLogic`

In addition, the `StakingLogic` contract has defined one key data structure `REWARDS` to keep track of the current reward distribution. It comes to our attention that the following two member fields in the above data structure are not used (and can then be safely removed): `totalClaimedReward` and `totalUnclaimedReward`.

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been resolved by the following commit: `186fd54`.

3.5 Improved Precision By Multiplication And Division Reordering

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

```
1052    function addReward(address reward, uint256 amt) external nonReentrant{
1053        require(msg.sender == parent, "Not parent");
1054        require(isRewardToken(reward), "Not reward");
1055        uint256 sh = amt.div(1000).mul(stakeDeployer(stakeDeployerAddress).rewardFee());
1056        if(reward == wETH){
1057        stakeInterface(parent).sendReward(reward, SD2(stakeDeployer(stakeDeployerAddress
            ).FEG()).BackingLogicAddress(), sh);
1058        amt -= sh;
1059        }
1060        _addPayout(reward, amt);
1061    }
```

Listing 3.5: StakingLogic::addReward()

In particular, we use the above `StakingLogic::addReward()` as an example. This routine is used to add new reward amount. We notice the calculation of the reward fee allotment (line 1055) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `sh = amt.mul(stakeDeployer(stakeDeployerAddress).rewardFee()).div(1000)`. Similarly, the calculation of `depositFee`, `withdrawFee` , `sacrificeLevel` and `boost` can be accordingly adjusted. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed by this commit: `d56cb0c`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

The staking support 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 setSacrificeEnabled(bool _bool) external onlyOwner{
        StakeLogics(stakeLogic).setSacrificeEnabled(_bool);
    }

    function setBurnWDFee(bool _bool) external onlyOwner{
        StakeLogics(stakeLogic).setBurnWDFee(_bool);
    }

    function setBoostBacking(bool _bool) external onlyOwner{
        StakeLogics(stakeLogic).setBoostBacking(_bool);
    }

    function setBoost(uint256 amt) external onlyOwner{
        StakeLogics(stakeLogic).setBoost(amt);
    }

    function setSyncLevel(address reward, uint256 amt) external onlyOwner{
```

```
        require(amt > 1e7, " 1e7");
        StakeLogics(stakeLogic).setSyncLevel(reward, amt);
    }
```

Listing 3.6: Example Privileged Operations in `StakingInterface`

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.

3.7 Bypassed Token Validation For Arbitrary Reward Addition

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Deployer`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `FEG Staking` support has a core `Deployer` contract that is used to create new pairs of `StakingLogic` and `StakingInterface` contracts. Also, the `Deployer` contract has an `isReward` mapping to record whether a given address is a reward token. Our analysis shows that this mapping may be manipulated to add other unexpected reward tokens.

```
    function setIsReward(address addy) external{
        require(isStake[msg.sender], "0L");
        isReward[addy] = true;
    }
```

Listing 3.7: `Deployer::setIsReward()`

In particular, we show above the related `setIsReward()` routine, which allows to add a new reward token. This routine is guarded with the requirement that the caller needs to be part of the `isStake` set. However, our analysis shows that it is possible to craft a token contract and then call the `createStake()` routine so that the crafted token contract is added into the `isStake` set. The only restrictions here are to ensure the token contract has a `sdOwner()` funtion to return the current caller as well as a no-op `setStakingAddress()` function. After that, it can be abused to add any token as the reward token.

```
    function createStake(address token, address reward, string memory name, string
        memory ticker) external returns (address SDS) {
        require(sdep(token).sdOwner() == msg.sender, "Only owner can create");
        require(!paused, "Creation paused");
        require(token != reward, "Token cannot be reward");
        require(!usedTicker[ticker], "Already used ticker");
        require(!usedName[name], "Already used name");
        require(!isStake[token], "Already created");
        ...
        isStake[token] = true;
        isStake[SDS] = true;
        isStake[stalog] = true;
        if(token != FEG){
        sdep(FEGStake).setRewardToken(token, 5e18);
        }
        tokenLogics[token].push(stalog);
        tint(SDS).setLogics(stalog);
        allSDS.push(SDS);
        allSSL.push(stalog);
        length += 1;
        ...
    }
```

Listing 3.8: `Deployer::createStake()`

Recommendation Revisit the above logic to ensure only intended reward tokens can be added.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `FEG Staking` support, which allows the stakers to benefit from the `FEG asset-backing` with the inherent `store-of-value` for an ever-rising baseline. As such, `FEG stakers` can earn `passive income` as all ecosystem-generated fees feed `FEG` staking rewards. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.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 CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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