SMART CONTRACT AUDIT REPORT

for

FEG Migrator

PeckShield

Prepared By: Xiaomi Huang

PeckShield

March 6, 2023

Document Properties

Client	FEG
Title	Smart Contract Audit Report
Target	FEG Migrator
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	March 6, 2023	Xuxian Jiang	Final Release
1.0-rc	February 29, 2023	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 `Migrator` 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 Migrator

`Migrator` is an `FEG`-related tool, which is used to migrate old `FEG` tokens to new `FEG` tokens. It accepts old `FEG` tokens that are directly held by users, or staked in the `FEGstake/FEGstakeV2` contracts. Moreover, it has a specific version for the `BSC` chain and the `Ethereum` chain. The basic information of the `Migrator` protocol is as follows:

Table 1.1: Basic Information of The `Migrator` Protocol

Item	Description
Name	FEG
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 6, 2023

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

- https://github.com/FEG-team/migrator.git (50cf3657)

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

- https://github.com/FEG-team/migrator.git (cb9f7f1e)

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

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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) 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's Migrator implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	0	
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 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 and 2 low-severity vulnerabilities.

Table 2.1: Key FEG Migrator Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improper amtClaimed Accounting in migrate()	Coding Practices	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	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.

PeckShield

3 | Detailed Results

3.1 Improper amtClaimed Accounting in migrate()

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: Migrator
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The Migrator provides users the ability to migrate their old FEG to the new one. It accepts the old FEG tokens that are directly held by users or staked in the FEGstake/FEGstakeV2 contracts. In order to facilitate users migration, it provides two ways for users to migrate their tokens. Firstly, users can migrate their FEG tokens on hand or staked in the FEGstake/FEGstakeV2 contracts separately. Secondly, users can migrate all their FEG tokens in one step.

It provides a state variable, i.e., amtClaimed, for users to check how many new FEG tokens they have claimed. While reviewing the one-step migration logic, we notice the amtClaimed is not updated. As a result, the amtClaimed variable cannot correctly reflect the amount of the new FEG tokens users have claimed. Based on this, it is suggested to update the amtClaimed in the migrate() routine.

```
    function migrate() external noReentrant lock{
        require(msg.sender == tx.origin, "no contract allowed");
        address user = msg.sender;
        uint256 toSend = 0;
        //balance
        if(IERC20(FEG).balanceOf(user) > 0){...}
        //Staking V_2
        if(IERC20(V_2).balanceOf(user) > 0){...}
        //Staking V_1
        if(!v1Claimed[user]){...}
        //checks if the person get's anything
        require(toSend > 0,"Nothing to migrate");
        require(IERC20(NEW_PAIR).transfer(user,toSend),"New token Transfer failed");
```

```
    }
```

Listing 3.1: `Migrator::migrate()`

Recommendation Properly update the `amtClaimed` state variable in the `migrate()` routine.

Status The issue has been fixed by these commits: `51a0779f` and `3d0390d2`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Migrator`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.2: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In current implementation, if we examine the `Migrator::saveLostTokens()` routine that is designed to rescue tokens from the contract to the `dev` account. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 120).

```
116   function saveLostTokens(address toSave) external { //added function to save any lost
          tokens
117     require(FEG != toSave,"Can't extract FEG");
118     require(msg.sender == dev, "You do not have permission");
119     uint256 toSend = IERC20(toSave).balanceOf(address(this));
120     require(IERC20(toSave).transfer(dev,toSend),"Extraction Transfer failed");
121   }
```

Listing 3.3: Migrator::saveLostTokens()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by these commits: `51a0779f` and `3d0390d2`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Migrator`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `Migrator` contract, there is a privileged account, i.e., `dev`, that can rescue tokens from the contract. Our analysis shows that the privileged account need to be scrutinized. In the following, we show the function potentially affected by the privilege of the `dev` account.

Specifically, the privileged function, i.e., `saveLostTokens()`, allows for the `dev` to rescue tokens from the contract. Note the tokens can be the `FEGstakeV2` LP tokens, which are transferred to the `Migrator` to migrate for the new `FEG`. However, the `FEGstakeV2` LP tokens can be rescued from the contract and further used by the `dev` to migrate for more new `FEG` tokens.

```
    function saveLostTokens(address toSave) external { //added function to save any lost
        tokens
      require(FEG != toSave,"Can't extract FEG");
```

```
        require(msg.sender == dev, "You do not have permission");
        uint256 toSend = IERC20(toSave).balanceOf(address(this));
        require(IERC20(toSave).transfer(dev,toSend),"Extraction Transfer failed");
    }
```

Listing 3.4: Migrator::saveLostTokens()

We understand the need of the privileged function for contract maintenance, but at the same time the extra power to the `dev` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `dev` 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 and the `FEGstakeV2` LP is not allowed to be rescued from the contract.

PeckShield

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `FEG Migrator` contract, which is design to migrate old `FEG` to a new one. It accepts the old `FEG` tokens that are directly held by users, or staked in the `FEGstake/FEGstakeV2` contracts. Moreover, it has a specific version for the `BSC` chain and the `Ethereum` chain. 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.

PeckShield

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.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.