SMART CONTRACT AUDIT REPORT

for

FLAMINGO

Prepared By: Shuxiao Wang

Hangzhou, China

Sep. 22, 2020

Document Properties

Client	Flamingo
Title	Smart Contract Audit Report
Target	Flamingo Staking
Version	1.0
Author	Edward Lo
Auditors	Edward Lo, Xudong Shao
Reviewed by	Shuxiao Wang, Jeff Liu, Chiachih Wu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Sep. 22, 2020	Edward Lo	Final Release
1.0-rc3	Sep. 21, 2020	Edward Lo	Release Candidate #3
1.0-rc2	Sep. 20, 2020	Edward Lo	Release Candidate #2
1.0-rc1	Sep. 20, 2020	Edward Lo	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Flamingo Staking** design document and related smart contract source code, we in the report outline 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 branch of Flamingo Staking 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 Flamingo Staking Contract

Flamingo is an interoperable, full-stack decentralized finance protocol built on the Neo blockchain. The Vault is Flamingo's one-stop asset manager, integrating asset staking/mining, and collateralized stable coin issuance. FLM tokens will be released, and users will receive FLM after staking whitelisted NEP-5 tokens (wrapped tokens and LP tokens) into the Vault.

The basic information of Flamingo Staking is as follows:

Table 1.1: Basic Information of Flamingo Staking

Item	Description
Issuer	Flamingo
Website	https://flamingo.finance/
Type	Neo Smart Contract
Platform	C#
Audit Method	Whitebox
Latest Audit Report	Sep. 22, 2020

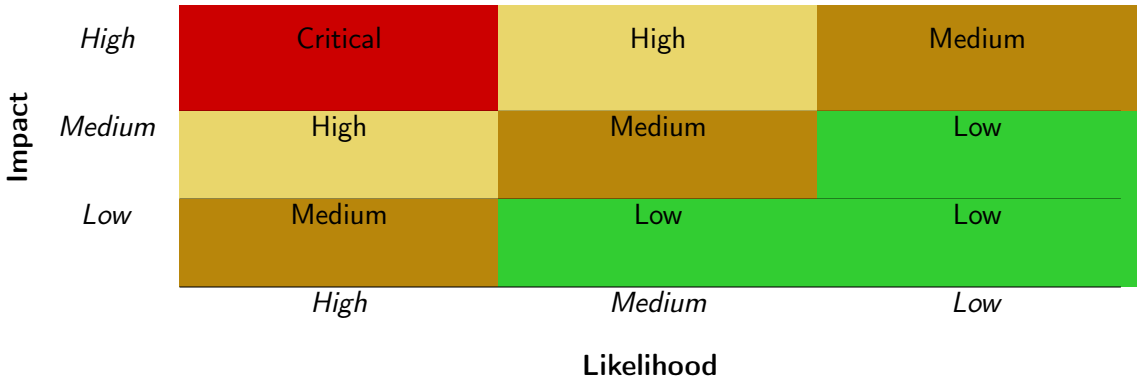
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/flamingo-finance/flamingo-contract-staking (2d35a3b)

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



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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Basic Coding Bugs Checks
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

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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 Flamingo Staking implementation. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the codebase. The purpose here is to not only statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool.

Severity	# of Findings	
Critical	2	■■
High	0	
Medium	1	■
Low	0	
Informational	6	■■■■■■
Total	9	

We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs. So far, we have 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 critical-severity vulnerabilities, 1 medium-severity vulnerability, and 6 informational recommendations.

Table 2.1: Key Flamingo Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	DivideByZeroException in `getuintprofit` Method	Coding Practices	Fixed
PVE-002	Informational	Missed `assetId` Validation in `getuintprofit` Method	Business Logics	Fixed
PVE-003	Informational	Violation of NEP-5 Standard in the FLM Contract	Business Logics	Fixed
PVE-004	Medium	Misused Sanity Check in the TransferFrom()	Coding Practices	Fixed
PVE-005	Informational	Behavior Discrepancy in the TransferFrom()	Business Logics	Fixed
PVE-006	Critical	Steal Tokens from Contract in Refund()	Business Logics	Fixed
PVE-007	Critical	Wrong Profit Calculation in ClaimFLM()	Coding Practices	Fixed
PVE-008	Informational	Missed Sanity Checks in ClaimFLM()	Coding Practices	Fixed
PVE-009	Informational	Missed Event in TransferFrom()	Coding Practices	Fixed

Please refer to Section 3 for details.

PeckShield

3 Detailed Results

3.1 DivideByZeroException in `getuintprofit` Method

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StakingReocrd.cs`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

In Flamingo Staking, the `getuintprofit` method allows the caller to query the staking profit of the specific asset. As shown in the code snippet below, the staking profit is calculated by dividing the `currentShareAmount` with `currentTotalStakingAmount` of the asset indexed by `assetId`. However, the division may cause the `DivideByZeroException` of NeoVM when `currentTotalStakingAmount` is zero, which could be avoided by ensuring `currentTotalStakingAmount > 0`.

```
85      else if (method == "getuintprofit")
86      {
87          var assetId = (byte[])args[0];
88
89          var currentTotalStakingAmount = GetCurrentTotalAmount(assetId);
90          var currentShareAmount = GetCurrentShareAmount(assetId);
91          return currentShareAmount / currentTotalStakingAmount;
92      }
```

Listing 3.1: StakingReocrd.cs

The `currentTotalStakingAmount` is derived from `GetCurrentTotalAmount(assetId)` which returns the amount of the asset (indexed by `assetId`) owned by the caller.

```
102     private static BigInteger GetCurrentTotalAmount(byte[] assetId)
103     {
104         var Params = new object[] { ExecutionEngine.ExecutingScriptHash };
105         BigInteger totalAmount = (BigInteger)((DyncCall)assetId.ToDelegate())("balanceOf
                ", Params);
```

```
106         return totalAmount;
107     }
```

Listing 3.2: GlobalRecordMethod.cs

Based on that, it is likely to get a zero `totalAmount` in line 105 when the caller passes an unknown `assetId` in or has zero balance of the specific asset, which leads to the divide-by-zero exception mentioned above.

Recommendation Ensure `currentTotalStakingAmount > 0` as follows:

```
85      else if (method == "getuintprofit")
86      {
87          var assetId = (byte[]) args[0];

89          var currentTotalStakingAmount = GetCurrentTotalAmount(assetId);
90          var currentShareAmount = GetCurrentShareAmount(assetId);
91          if ( currentTotalStakingAmount == 0 ) {
92              return 0;
93          }
94          return currentShareAmount / currentTotalStakingAmount;
95      }
```

Listing 3.3: StakingReocrd.cs

Status The issue has been fixed by this commit: `e6fa031`.

3.2 Missed `assetId` Validation in `getuintprofit` Method

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StakingReocrd.cs`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

As analyzed in Section 3.1, the `getuintprofit` computes the staking profit of the asset indexed by `assetId`. However, the `assetId` is retrieved by `args` directly without any validation logic. This leads to uncertain behaviors as users may accidentally pass in any script which declared the balanceOf ABI.

```
85          else if (method == "getuintprofit")
86          {
87              var assetId = (byte[]) args[0];

89              var currentTotalStakingAmount = GetCurrentTotalAmount(assetId);
90              var currentShareAmount = GetCurrentShareAmount(assetId);
```

```
91              return currentShareAmount / currentTotalStakingAmount;
92          }
```

Listing 3.4: StakingReocrd.cs

Recommendation Add sanity check as follows:

```
85          else if (method == "getuintprofit")
86          {
87              var assetId = (byte[])args[0];
88              if (!IsInWhiteList(assetId) || assetId.Length != 20) return 0;

90              var currentTotalStakingAmount = GetCurrentTotalAmount(assetId);
91              var currentShareAmount = GetCurrentShareAmount(assetId);
92              return currentShareAmount / currentTotalStakingAmount;
93          }
```

Listing 3.5: StakingReocrd.cs

Status The issue has been fixed by this commit: `e6fa031`.

3.3 Violation of NEP-5 Standard in the FLM Contract

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FLM.cs`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

The NEP-5 proposal outlines a token standard for the Neo blockchain that will provide systems with a generalized interaction mechanism for tokenized Smart Contracts. Different from UTXO, the NEP5 assets are recorded in the contract storage area, through updating account balance in the storage area, to complete the transaction. The FLM contract implements the `Transfer()` function following the NEP-5 standard. However, the current implementation has two minor issues which violate the NEP-5 standard.

Specifically, when the `from` and `to` addresses are the same, the current implementation fails to fire the event but simply return `true` in line 105. In addition, when the `to` address is not payable, the `Transfer()` method should return `false` but the current implementation somehow has those error handling code disabled (line 107-110).

```
96          public static bool Transfer(byte[] from, byte[] to, BigInteger amt, byte[]
                callingScript)
97          {
```

```
98              // strictly follow the protocol https://github.com/neo-project/proposals/
                    blob/master/nep-5.mediawiki#transfer
99              assert(from.Length == 20 && to.Length == 20 , "transfer: invalid from or to,
                     from-".AsByteArray().Concat(from).Concat(" and to-".AsByteArray()).
                    Concat(to).AsString());
100             assert(Runtime.CheckWitness(from) || from.Equals(callingScript), "transfer:
                    CheckWitness failed, from-".AsByteArray().Concat(from).AsString());
101             assert(amt >= 0, "transfer: invalid amount-".AsByteArray().Concat(amt.
                    ToByteArray()).AsString());

103             if (from.Equals(to))
104             {
105                 return true;
106             }
107             //if (!Blockchain.GetContract(to).IsPayable)
108             //{
109             //    return false;
110             //}
```

Listing 3.6: FLM.cs

Recommendation Fix the two violations as follows:

```
96          public static bool Transfer(byte[] from, byte[] to, BigInteger amt, byte[]
                callingScript)
97          {
98              // strictly follow the protocol https://github.com/neo-project/proposals/
                    blob/master/nep-5.mediawiki#transfer
99              assert(from.Length == 20 && to.Length == 20 , "transfer: invalid from or to,
                     from-".AsByteArray().Concat(from).Concat(" and to-".AsByteArray()).
                    Concat(to).AsString());
100             assert(Runtime.CheckWitness(from) || from.Equals(callingScript), "transfer:
                    CheckWitness failed, from-".AsByteArray().Concat(from).AsString());
101             assert(amt >= 0, "transfer: invalid amount-".AsByteArray().Concat(amt.
                    ToByteArray()).AsString());

103             if (from.Equals(to))
104             {
105                 TransferEvent(from, to, amt);
106                 return true;
107             }
108             if (!Blockchain.GetContract(to).IsPayable)
109             {
110                 return false;
111             }
```

Listing 3.7: FLM.cs

Status The issue has been fixed by this commit: 28c5be0.

3.4 Misused Sanity Check in the TransferFrom()

- ID: PVE-004
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: FLM.cs
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

Besides the Transfer() method described in Section 3.3, the FLM contract also implements the TransferFrom() method which has lots of use cases in DeFi scenario. In particular, the TransferFrom() method allows the owner to approve the spender to transfer owner's assets to the receiver. However, there is a misused sanity check in FLM's TransferFrom() implementation such that the owner could spend the tokens which are already approved to the spender.

```
148         public static bool TransferFrom(byte[] spender, byte[] owner, byte[] receiver,
                BigInteger amt, byte[] callingScript)
149         {
150             assert(spender.Length == 20 && owner.Length == 20 && receiver.Length == 20,
                    "transferFrom: invalid spender or owner or receiver, spender-".
                    AsByteArray().Concat(spender).Concat(", owner-".AsByteArray()).Concat(
                    owner).Concat(" and receiver-".AsByteArray()).Concat(receiver).AsString
                    ());
151             assert(amt > 0, "transferFrom: invalid amount-".AsByteArray().Concat(amt.
                    ToByteArray()).AsString());
152           assert(Runtime.CheckWitness(spender) || owner.Equals(callingScript), "
                  transferFrom: CheckWitness failed, spender-".AsByteArray().Concat(spender)
                  .AsString());

154             if (spender.Equals(owner) || owner.Equals(receiver))
155             {
156                 return true;
157             }
```

Listing 3.8: FLM.cs

Specifically, as shown in the code snippet, the owner.Equals(callingScript) in line 152 would be true when the owner calls the TransferFrom() method. This allows the owner to spend the already approved tokens.

Recommendation Remove the owner.Equals(callingScript) check in line 152.

```
148         public static bool TransferFrom(byte[] spender, byte[] owner, byte[] receiver,
                BigInteger amt, byte[] callingScript)
149         {
150             assert(spender.Length == 20 && owner.Length == 20 && receiver.Length == 20,
                    "transferFrom: invalid spender or owner or receiver, spender-".
```

```
                AsByteArray().Concat(spender).Concat(", owner-".AsByteArray()).Concat(
                owner).Concat(" and receiver-".AsByteArray()).Concat(receiver).AsString
                ());
151             assert(amt > 0, "transferFrom: invalid amount-".AsByteArray().Concat(amt.
                    ToByteArray()).AsString());
152         assert(Runtime.CheckWitness(spender), "transferFrom: CheckWitness failed,
                spender-".AsByteArray().Concat(spender).AsString());

154         if (spender.Equals(owner) || owner.Equals(receiver))
155           {
156               return true;
157           }
```

Listing 3.9: FLM.cs

Status The issue has been fixed by this commit: `11fc7d8`.

3.5 Behavior Discrepancy in the TransferFrom()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FLM.cs`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

As described in Section 3.4, the `TransferFrom()` method allows the `spender` to move tokens from `owner` to `receiver`. However, there is a behavior discrepancy in FLM's `TransferFrom()` implementation. Specifically, when `owner` equals `spender`, `TransferFrom()` simply returns `true` (line 157 below) due to the fact that the `owner` has no reason to approve herself as the `spender` to move her own tokens.

```
148         public static bool TransferFrom(byte[] spender, byte[] owner, byte[] receiver,
                BigInteger amt, byte[] callingScript)
149         {
150             assert(spender.Length == 20 && owner.Length == 20 && receiver.Length == 20,
                    "transferFrom: invalid spender or owner or receiver, spender-".
                    AsByteArray().Concat(spender).Concat(", owner-".AsByteArray()).Concat(
                    owner).Concat(" and receiver-".AsByteArray()).Concat(receiver).AsString
                    ());
151             assert(amt > 0, "transferFrom: invalid amount-".AsByteArray().Concat(amt.
                    ToByteArray()).AsString());
152           assert(Runtime.CheckWitness(spender) || owner.Equals(callingScript), "
                  transferFrom: CheckWitness failed, spender-".AsByteArray().Concat(spender)
                  .AsString());

154           if (spender.Equals(owner) || owner.Equals(receiver))
```

```
155             {
156                 return true;
157             }
```

Listing 3.10: FLM.cs

But, in the `Approve()` function, the `owner` is allowed to approve herself as the `spender`, which is inconsistent to the logic in `TransferFrom()`.

```
132         public static bool Approve(byte[] owner, byte[] spender, BigInteger amt, byte[]
                callingScript)
133         {
134             assert(owner.Length == 20 && spender.Length == 20, "approve: invalid owner
                    or spender, owner-".AsByteArray().Concat(owner).Concat("and spender-".
                    AsByteArray()).Concat(spender).AsString());
135             assert(amt > 0, "approve: invalid amount-".AsByteArray().Concat(amt.
                    ToByteArray()).AsString());
136             assert(Runtime.CheckWitness(owner) || owner.Equals(callingScript), "approve:
                    CheckWitness failed, owner-".AsByteArray().Concat(owner).AsString());

138             Storage.Put(AllowancePrefix.Concat(owner).Concat(spender), amt);
139             ApproveEvent(owner, spender, amt);
140             return true;
141         }
```

Listing 3.11: FLM.cs

Since `Approve()` and `TransferFrom()` are usually used at the same time, we suggest to have consistent logic implemented in these two functions.

Recommendation Make `Approve()` returns when `spender.Equals(owner)`.

```
132         public static bool Approve(byte[] owner, byte[] spender, BigInteger amt, byte[]
                callingScript)
133         {
134             assert(owner.Length == 20 && spender.Length == 20, "approve: invalid owner
                    or spender, owner-".AsByteArray().Concat(owner).Concat("and spender-".
                    AsByteArray()).Concat(spender).AsStri    ng());
135             assert(amt > 0, "approve: invalid amount-".AsByteArray().Concat(amt.
                    ToByteArray()).AsString());
136         assert(Runtime.CheckWitness(owner) || owner.Equals(callingScript), "approve:
                CheckWitness failed, owner-".AsByteArray().Concat(owner).AsString());

138             if (spender.Equals(owner))
139             {
140                 return true;
141             }

143             Storage.Put(AllowancePrefix.Concat(owner).Concat(spender), amt);
144             ApproveEvent(owner, spender, amt);
145             return true;
146         }
```

Listing 3.12: FLM.cs

Status The issue has been fixed by this commit: `a7d19a1`.

3.6 Steal Tokens from Contract in Refund()

- ID: PVE-006
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `StakingReocrd.cs`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

In Flamingo Staking, the `Staking` function allows users to stake their tokens for making profits. When an user tends to get back her staked assets, the `Refund()` function needs to be called. While reviewing the staking/refunding business logic, we came across a critical issue which allows a bad actor to steal tokens from the staking pool. Specifically, as shown in the code snippet below, the `Staking()` function retrieves the old staking records from the storage (line 153) and updates the records with the newly staked `amount` (line 162).

```
112 public static bool Staking(byte[] fromAddress, BigInteger amount, byte[] assetId)
113 {
114     if (!IsInWhiteList(assetId) ||assetId.Length != 20 ||CheckWhetherSelf(fromAddress))
        return false; //throw exception when release
115     object[] Params = new object[]
116     {
117         fromAddress,
118         ExecutionEngine.ExecutingScriptHash,
119         amount
120     };
121     BigInteger currentTimeStamp = GetCurrentTimeStamp();
122     if (!checkIfStakingStart(currentTimeStamp)) return false;
123     if (!(bool)((DyncCall)assetId.ToDelegate())("transfer", Params)) return false; //
        throw exception when release
124     byte[] key = assetId.Concat(fromAddress);
125     var result = Storage.Get(key);
126     BigInteger currentProfit = 0;
127     UpdateStackRecord(assetId, currentTimeStamp);
128     if (result.Length != 0)
129     {
130         StakingReocrd stakingRecord = (StakingReocrd)result.Deserialize();
131         currentProfit = SettleProfit(stakingRecord.timeStamp, stakingRecord.amount,
            assetId) + stakingRecord.Profit;
132         amount += stakingRecord.amount;
133     }
134     SaveUserStaking(fromAddress, amount, assetId, currentTimeStamp, currentProfit, key);
135     return true;
```

```
136 }
```

Listing 3.13: StakingReocrd.cs

On the other hand, when the user `Refund()` the staked tokens, the function transfers out `amount` of tokens to `fromAddress` in line 151. After that, the `stakingRecord.amount` is checked in line 157. But even when `stakingRecord.amount < amount`, the function returns `false` instead of reverts the transaction. Those assets indexed by `assetId` are transfered out regardless of the `stakingRecord.amount` check. It means a bad actor could `Staking()` a small amount of tokens (for bypassing the check in line 146) and drain the staking pool with `Refund()`.

```
138 public static bool Refund(byte[] fromAddress, BigInteger amount, byte[] assetId)
139 {
140     //
141     if (!Runtime.CheckWitness(fromAddress)) return false;
142     BigInteger currentTimeStamp = GetCurrentTimeStamp();
143     if (!checkIfRefundStart(currentTimeStamp)) return false;
144     byte[] key = assetId.Concat(fromAddress);
145     var result = Storage.Get(key);
146     if (result.Length == 0) return false;
147     StakingReocrd stakingRecord = (StakingReocrd)result.Deserialize();
148     //
149     object[] Params = new object[]
150     {
151         ExecutionEngine.ExecutingScriptHash,
152         fromAddress,
153         amount
154     };
155     DyncCall nep5Contract = (DyncCall)assetId.ToDelegate();
156     if (!(bool)nep5Contract("transfer", Params)) return false; //throw exception when
        release
157     if (stakingRecord.amount < amount ||!(stakingRecord.fromAddress.Equals(fromAddress))
        ||!(stakingRecord.assetId.Equals(assetId)))
158     {
159         return false;
160     }
```

Listing 3.14: StakingReocrd.cs

Recommendation Validate the `stakingRecord.amount` before transfering tokens out.

```
138 public static bool Refund(byte[] fromAddress, BigInteger amount, byte[] assetId)
139 {
140     //
141     if (!Runtime.CheckWitness(fromAddress)) return false;
142     BigInteger currentTimeStamp = GetCurrentTimeStamp();
143     if (!checkIfRefundStart(currentTimeStamp)) return false;
144     byte[] key = assetId.Concat(fromAddress);
145     var result = Storage.Get(key);
146     if (result.Length == 0) return false;
147     StakingReocrd stakingRecord = (StakingReocrd)result.Deserialize();
```

```
149     if (stakingRecord.amount < amount ||!(stakingRecord.fromAddress.Equals(fromAddress))
            ||!(stakingRecord.assetId.Equals(assetId)))
150     {
151         return false;
152     }


155     //
156     object[] Params = new object[]
157     {
158         ExecutionEngine.ExecutingScriptHash,
159         fromAddress,
160         amount
161     };
162     DyncCall nep5Contract = (DyncCall)assetId.ToDelegate();
163     if (!(bool)nep5Contract("transfer", Params)) return false; //throw exception when
            release
```

Listing 3.15: StakingReocrd.cs

Status The issue has been fixed by this commit: `a7d19a1`.

3.7 Wrong Profit Calculation in ClaimFLM()

- ID: PVE-007
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `StakingRecord`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

Similar to the staking/refunding logic described in Section 3.6, users can also choose to claim FLM tokens with `ClaimFLM()` after `Staking()` assets. While reviewing the FLM generating logic, we identified another critical issue which may destroy the FLM tokenomics.

```
184         public static bool ClaimFLM(byte[] fromAddress, byte[] assetId, byte[]
                callingScript)
185         {
186             if (!Runtime.CheckWitness(fromAddress)) return false;
187             byte[] key = assetId.Concat(fromAddress);
188             StakingReocrd stakingReocrd = (StakingReocrd)Storage.Get(key).Deserialize();
189             if (!stakingReocrd.fromAddress.Equals(fromAddress))
190             {
191                 return false;
192             }
193             UpdateStackRecord(assetId, GetCurrentTimeStamp());
```

```
194             BigInteger newProfit = SettleProfit(stakingReocrd.timeStamp, stakingReocrd.
                    amount, assetId);
195             var profitAmount = stakingReocrd.Profit + newProfit;
196             SaveUserStaking(fromAddress, stakingReocrd.amount, stakingReocrd.assetId,
                    Blockchain.GetHeight(), 0, key);
197             if (!MintFLM(fromAddress, profitAmount, callingScript))
198             {
199                 return false;
200             }
201             return true;
202         }
```

Listing 3.16: StakingRecord.cs

Specifically, `ClaimFLM()` retrieves `stakingReocrd.timeStamp` from the storage (line 188) and invokes `SettleProfit()` with the timestamp to calculate the `newProfit` (line 194). The `newProfit` is later used to update the `profitAmount` which is the amount to mint FLM tokens (line 195). However, in line 196, the staking record is wrongly updated with the `Blockchain.GetHeight()` instead of the timesteamp. This leads to the wrong profit calculation as decribed in the following:

```
219 public static BigInteger SettleProfit(BigInteger recordTimeStamp, BigInteger amount,
        byte[] assetId)
220 {
221     BigInteger MinusProfit = GetHistoryUintStackProfitSum(assetId, recordTimeStamp);
222     BigInteger SumProfit = GetHistoryUintStackProfitSum(assetId, GetCurrentTimeStamp());
223     BigInteger currentProfit = (SumProfit - MinusProfit) * amount;
224     return currentProfit;
225 }
```

Listing 3.17: StakingRecord.cs

Inside `SettleProfit()`, `GetHistoryUintStackProfitSum()` is called to calculate the `MinusProfit` based on the history as the functon name suggested. Since the storage is not correctly updated, `GetHistoryUintStackProfitSum()` always returns 0 (line 20 in code snippet below). As a result, the `currentProfit` returned by `SettleProfit()` would include all the already claimed FLM tokens, which is not expected.

```
14 private static BigInteger GetHistoryUintStackProfitSum(byte[] assetId, BigInteger
       TimeStamp)
15 {
16     byte[] key = _historyUintStackProfitSum.Concat(assetId.Concat(TimeStamp.AsByteArray
           ()));
17     var result = Storage.Get(key);
18     if (result.Length == 0)
19     {
20         return 0;
21     }
22     else
23     {
24         return result.ToBigInteger();
25     }
```

```
26 }
```

Listing 3.18: GlobalRecordMethod.cs

Recommendation Use correct API for calculation, i.e., `GetCurrentTimeStamp()`.

Status The issue has been fixed by this commit: `3935a4e`.

3.8 Missed Sanity Checks in ClaimFLM()

- ID: PVE-008
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: `StakingRecord.cs`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

As decribed in Section 3.7, the `ClaimFLM()` allows the users who stake assets to claim the rewarded FLM tokens. While reviewing the `ClaimFLM()` function, we identified a missed sanity check which may lead to an exception.

```
184         public static bool ClaimFLM(byte[] fromAddress, byte[] assetId, byte[]
                callingScript)
185         {
186             if (!Runtime.CheckWitness(fromAddress)) return false;
187             byte[] key = assetId.Concat(fromAddress);
188             StakingReocrd stakingReocrd = (StakingReocrd)Storage.Get(key).Deserialize();
189             if (!stakingReocrd.fromAddress.Equals(fromAddress))
190             {
191                 return false;
192             }
193             UpdateStackRecord(assetId, GetCurrentTimeStamp());
194             BigInteger newProfit = SettleProfit(stakingReocrd.timeStamp, stakingReocrd.
                    amount, assetId);
195             var profitAmount = stakingReocrd.Profit + newProfit;
196             SaveUserStaking(fromAddress, stakingReocrd.amount, stakingReocrd.assetId,
                    Blockchain.GetHeight(), 0, key);
197             if (!MintFLM(fromAddress, profitAmount, callingScript))
198             {
199                 return false;
200             }
201             return true;
202         }
```

Listing 3.19: StakingRecord.cs

Specifically, ClaimFLM() derives the key to retrieve stakingReocrd from the storage by combining assetId and fromAddress (line 187). However, ClaimFLM() fails to validate the result retrieved from the storage but Deserialize() the result directly. If ClaimFLM() is called with nonexistent assetId and fromAddress, Deserialize would handle null record and cause an exception. Although the exception will be caught by NeoVM, adding a sanity check to prevent it would be a better solution.

Recommendation Check the retrieved stakingReocrd before Deserialize().

Status The issue has been fixed by this commit: 3b07ce7.

3.9 Missed Event in TransferFrom()

- ID: PVE-009
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: FLM.cs
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.3 and Section 3.4, the FLM contract implements Transfer() and TransferFrom(). According to the NEP-5 proposal, Transfer() should allow zero amount transfers with corresponding event emitted. As shown in the code snippet below, the assertion in line 101 ensures amt >= 0, which means a Transfer() call with amt = 0 could pass.

```
96  public static bool Transfer(byte[] from, byte[] to, BigInteger amt, byte[] callingScript
        )
97  {
98      // strictly follow the protocol https://github.com/neo-project/proposals/blob/master
            /nep-5.mediawiki#transfer
99      assert(from.Length == 20 && to.Length == 20 , "transfer: invalid from or to, from-".
            AsByteArray().Concat(from).Concat(" and to-".AsByteArray()).Concat(to).AsString
            ());
100     assert(Runtime.CheckWitness(from) ||from.Equals(callingScript), "transfer:
            CheckWitness failed, from-".AsByteArray().Concat(from).AsString());
101     assert(amt >= 0, "transfer: invalid amount-".AsByteArray().Concat(amt.ToByteArray())
            .AsString());
102
103     if (from.Equals(to))
104     {
105         return true;
106     }
107     //if (!Blockchain.GetContract(to).IsPayable)
108     //{
109     //    return false;
```

```
110     //}
```

Listing 3.20: FLM.cs

However, when we review the `TransferFrom()` implementation, we identified that `TransferFrom()` rejects zero amount transfers by `assert(amt > 0)`.

```
148 public static bool TransferFrom(byte[] spender, byte[] owner, byte[] receiver,
        BigInteger amt, byte[] callingScript)
149 {
150     assert(spender.Length == 20 && owner.Length == 20 && receiver.Length == 20, "
            transferFrom: invalid spender or owner or receiver, spender-".AsByteArray().
            Concat(spender).Concat(", owner-".AsByteArray()).Concat(owner).Concat(" and
            receiver-".AsByteArray()).Concat(receiver).AsString());
151     assert(amt > 0, "transferFrom: invalid amount-".AsByteArray().Concat(amt.ToByteArray
            ()).AsString());
152     assert(Runtime.CheckWitness(spender) ||owner.Equals(callingScript), "transferFrom:
            CheckWitness failed, spender-".AsByteArray().Concat(spender).AsString());
```

Listing 3.21: FLM.cs

Although `TransferFrom()` is not defined in NEP-5, we suggest to have consistent logic implemented in these two functions.

Recommendation Trigger transfer event even when `amt = 0` in `TransferFrom`.

```
148 public static bool TransferFrom(byte[] spender, byte[] owner, byte[] receiver,
        BigInteger amt, byte[] callingScript)
149 {
150     assert(spender.Length == 20 && owner.Length == 20 && receiver.Length == 20, "
            transferFrom: invalid spender or owner or receiver, spender-".AsByteArray().
            Concat(spender).Concat(", owner-".AsByteArray()).Concat(owner).Concat(" and
            receiver-".AsByteArray()).Concat(receiver).AsString());
151     assert(amt >= 0, "transferFrom: invalid amount-".AsByteArray().Concat(amt.
            ToByteArray()).AsString());
152     assert(Runtime.CheckWitness(spender) ||owner.Equals(callingScript), "transferFrom:
            CheckWitness failed, spender-".AsByteArray().Concat(spender).AsString());
```

Listing 3.22: FLM.cs

Status The issue has been fixed by this commit: `a7d19a1`.

3.10 Other Suggestions

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.

4 Conclusion

In this audit, we thoroughly analyzed the Flamingo Staking design and implementation. Flamingo Staking contract helps users to receive FLM tokens by staking whitelisted wrapped tokens or LP tokens. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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