SMART CONTRACT AUDIT REPORT

for

FLAMINGO

Prepared By: Shuxiao Wang

Hangzhou, China

Sep. 30, 2020

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	Sep. 30, 2020	Edward Lo	Final Release Version
1.0-rc1	Sep. 25, 2020	Edward Lo	Release Candidate #2
1.0-rc2	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 design document and related source code of the **Flamingo Swap** smart contract, we in this 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 Swap smart contract 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 Swap Contract

Flamingo Swap is Flamingo's on-chain Auto Market Maker (AMM), providing liquidity to wrapped assets, FLM, and other NEP-5 based tokens. Swap adopts the Constant Product Market Maker (CPMM) model, which was popularized in many AMM-based DEXs, such as Uniswap. CPMMs are based on a function that establishes a range of prices for two tokens according to the available quantities (liquidity) of each token. Within Swap, users can trade token pairs (included in a whitelist at the early stage) or provide liquidity to a chosen liquidity pool by depositing tokens to provide equal liquidity on both sides of the trading pair.

The basic information of Flamingo Swap smart contract is as follows:

Table 1.1: Basic Information of Flamingo Swap smart contract

Item	Description
Issuer	Flamingo
Website	https://flamingo.finance/
Type	Neo Smart Contract
Platform	C#
Audit Method	Whitebox
Latest Audit Report	Sep. 30, 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-swap (fceda2a)

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.

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
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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 Swap 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.

Table 2.1: The Severity of Our Findings

Severity	# of Findings	
Critical	0	
High	3	■ ■ ■
Medium	0	
Low	1	■
Informational	2	■ ■
Total	6	■ ■ ■ ■ ■ ■

2.2 Key Findings

Overall, the smart contract implementation can be improved because of the existence of 3 high vulnerabilities, which requires an urgent fix-up, 2 low severity vulnerabilities, and 1 informational recommendations, as shown in Table 2.2.

Table 2.2: Key Audit Findings

ID	Severity	Title	Type	Status
PVE-001	Low	Missed Sanity Check on estimatedA	Improper Check for Unusual or Exceptional Conditions	Fixed
PVE-002	High	Lack of Sanity Check in FlamingoSwapRouterContract's Upgrade()	Business Logics	Fixed
PVE-003	High	Lack of Sanity Check in FlamingoSwapPairContract's Upgrade()	Business Logics	Fixed
PVE-004	High	Lack of Sanity Check in FlamingoSwapFactoryContract's Upgrade()	Business Logics	Fixed
PVE-005	Info.	Violation of NEP-5 Standard in the Pair Contract	Business Logics	Fixed
PVE-006	Low	Missed Permission Checks in Mint(), Burn(), and Swap()	Business Logics	Fixed

Please refer to Chapter 3 for details.

3 Detailed Results

3.1 Missed Sanity Check on estimatedA

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FlamingoSwapRouterContract.cs`
- Category: Improper Check for Unusual or Exceptional Conditions [2]
- CWE subcategory: CWE-703 [1]

Description

In Flamingo Swap, the `FlamingoSwapRouterContract` contract provides interfaces for users to add/remove liquidity and swap tokens. The function `AddLiquidity()` can add tokens to the pool and mint liquidity tokens to the user. During the adding liquidity process, the min/max amounts of the pair of tokens are passed into `AddLiquidity()` function. As shown in the code snippets below, the `Quote()` function is invoked (line 101 and line 110) to estimate the amounts to be transferred into the `pairContract` (i.e., `amountA` and `amountB`). When `Quote()` returns `estimatedB`, line 102 and line 104 ensure that the estimation is in the valid range. However, the checks against `estimatedA` is not applied completely, leading to a business logic error which allows `estimatedA` > `amountADesired`.

```
82 public static BigInteger[] AddLiquidity(byte[] sender, byte[] tokenA, byte[] tokenB,
       BigInteger amountADesired, BigInteger amountBDesired, BigInteger amountAMin,
       BigInteger amountBMin, BigInteger deadLine)
83 {
84     Assert(Runtime.CheckWitness(sender), "Forbidden");
85 
86     Assert((BigInteger) Runtime.Time <= deadLine, "Exceeded the deadline");
87 
88 
89     var reserves = GetReserves(tokenA, tokenB);
90     var reserveA = reserves[0];
91     var reserveB = reserves[1];
92     BigInteger amountA = 0;
93     BigInteger amountB = 0;
94     if (reserveA == 0 && reserveB == 0)
```

```
    {
        amountA = amountADesired;
        amountB = amountBDesired;
    }
    else
    {
        var estimatedB = Quote(amountADesired, reserveA, reserveB);
        if (estimatedB <= amountBDesired)
        {
            Assert(estimatedB >= amountBMin, "Insufficient B Amount");
            amountA = amountADesired;
            amountB = estimatedB;
        }
        else
        {
            var estimatedA = Quote(amountBDesired, reserveB, reserveA);
            Assert(estimatedA >= amountAMin, "Insufficient A Amount");
            amountA = estimatedA;
            amountB = amountBDesired;
        }
    }

    var pairContract = GetExchangePairWithAssert(tokenA, tokenB);

    SafeTransfer(tokenA, sender, pairContract, amountA);
    SafeTransfer(tokenB, sender, pairContract, amountB);

    var liquidity = pairContract.DynamicMint(sender);//+0.03gas
    //var liquidity = ((Func<string, object[], BigInteger>)pairContract.ToDelegate())("
        mint", new object[] { sender });
    return new BigInteger[] { amountA.ToBigInt(), amountB.ToBigInt(), liquidity };
}
```

Listing 3.1: FlamingoSwapRouterContract.cs

In particular, since `amountADesired` and `amountBDesired` are the maximum amount of tokens that can be transferred in, the final input token amounts `amountA` and `amountB` should be less than `amountADesired` and `amountBDesired`. Based on that, in the case that `estimatedB` is larger than `amountBDesired`, `estimatedA` should be checked against `amountADesired` to make the assumption hold.

Recommendation Add sanity checks on `estimatedA` to ensure that it is less than or equal to `amountADesired`.

```
public static BigInteger[] AddLiquidity(byte[] sender, byte[] tokenA, byte[] tokenB,
    BigInteger amountADesired, BigInteger amountBDesired, BigInteger amountAMin,
    BigInteger amountBMin, BigInteger deadLine)
{
    Assert(Runtime.CheckWitness(sender), "Forbidden");

    Assert((BigInteger) Runtime.Time <= deadLine, "Exceeded the deadline");

```

```

    var reserves = GetReserves(tokenA, tokenB);
    var reserveA = reserves[0];
    var reserveB = reserves[1];
    BigInteger amountA = 0;
    BigInteger amountB = 0;
    if (reserveA == 0 && reserveB == 0)
    {
        amountA = amountADesired;
        amountB = amountBDesired;
    }
    else
    {
        var estimatedB = Quote(amountADesired, reserveA, reserveB);
        if (estimatedB <= amountBDesired)
        {
            Assert(estimatedB >= amountBMin, "Insufficient B Amount");
            amountA = amountADesired;
            amountB = estimatedB;
        }
        else
        {
            var estimatedA = Quote(amountBDesired, reserveB, reserveA);
            Assert(estimatedA <= amountADesired, "Insufficient A Amount");
            Assert(estimatedA >= amountAMin, "Insufficient A Amount");
            amountA = estimatedA;
            amountB = amountBDesired;
        }  //var liquidity = ((Func<string, object[], BigInteger>)pairContract.
            ToDelegate())("mint", new object[] { sender });
    }

    var pairContract = GetExchangePairWithAssert(tokenA, tokenB);

    SafeTransfer(tokenA, sender, pairContract, amountA);
    SafeTransfer(tokenB, sender, pairContract, amountB);

    var liquidity = pairContract.DynamicMint(sender);//+0.03gas

    return new BigInteger[] { amountA.ToBigInt(), amountB.ToBigInt(), liquidity };
}
```

Listing 3.2: FlamingoSwapRouterContract.cs

Status The issue has been fixed by this commit: `04eb648`.

3.2 Front-Running Risk of Upgrading Contract in FlamingoSwapRouterContract

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: FlamingoSwapRouterContract.Admin.cs
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [3]

Description

As a standard contract migration implementation, the Router contract has a Upgrade() method which allows the admin to trigger a contract upgrade with the provided newScript. However, the deployment of the new contract could be front-run'ed such that the states on the original contract cannot be migrated.

```
public static byte[] Upgrade(byte[] newScript, byte[] paramList, byte returnType,
    ContractPropertyState cps, string name, string version, string author, string email,
     string description)
{
    Assert(Runtime.CheckWitness(GetAdmin()), "upgrade: CheckWitness failed!");

    byte[] newContractHash = Hash160(newScript);

    Contract newContract = Contract.Migrate(newScript, paramList, returnType, cps, name,
         version, author, email, description);
    Runtime.Notify("upgrade", ExecutionEngine.ExecutingScriptHash, newContractHash);
    return newContractHash;
}
```

Listing 3.3: FlamingoSwapRouterContract.Admin.cs

Specicailly, the Upgrade() function calls Contract.Migrate() to upgrade the contract. Contract.Migrate() migrates everything in the persistent storage of the current contract to the new contract when executed. In particular, the Migrate() method only transfers the contract storages when the target contract has not been deployed yet. Based on that, if the new contract has been deployed already, Router contract's states would not be migrated.

Recommendation Check whether the contract already exists before calling Contract.Migrate().

Status The issue has been fixed by this commit: 04eb648.

3.3 Lack of Sanity Check in FlamingoSwapPairContract's Upgrade()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: FlamingoSwapFactoryContract.Admin.cs
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [3]

Description

As a standard contract migration implementation, the Pair contract has a Upgrade() method which allows the admin to trigger a contract upgrade with the provided newScript. However, the deployment of the new contract could be front-run'ed such that the states on the original contract cannot be migrated.

```
public static byte[] Upgrade(byte[] newScript, byte[] paramList, byte returnType,
    ContractPropertyState cps, string name, string version, string author, string email,
     string description)
{
    Assert(Runtime.CheckWitness(GetAdmin()), "upgrade: CheckWitness failed!");

    var me = ExecutionEngine.ExecutingScriptHash;
    byte[] newContractHash = Hash160(newScript);

    var r = ReservePair;
    SafeTransfer(Token0, me, newContractHash, r.Reserve0);
    SafeTransfer(Token1, me, newContractHash, r.Reserve1);

    Contract newContract = Contract.Migrate(newScript, paramList, returnType, cps, name,
         version, author, email, description);

    Runtime.Notify("upgrade", ExecutionEngine.ExecutingScriptHash, newContractHash);
    return newContractHash;
}
```

Listing 3.4: FlamingoSwapPairContract.Admin.cs

Specicailly, the Upgrade() function calls Contract.Migrate() to upgrade the contract. Contract.Migrate() migrates everything in the persistent storage of the current contract to the new contract when executed. In particular, the Migrate() method only transfers the contract storages when the target contract has not been deployed yet. Based on that, if the new contract has been deployed already, Router contract's states would not be migrated.

Recommendation Check whether the contract already exists before calling `Contract.Migrate()`. And transfer the tokens after the contract migration has succeeded.

Status The issue has been fixed by this commit: `04eb648`.

3.4 Lack of Sanity Check in FlamingoSwapFactoryContract's Upgrade()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `FlamingoSwapFactoryContract.Admin.cs`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [3]

Description

As a standard contract migration implementation, the `Factory` contract has a `Upgrade()` method which allows the admin to trigger a contract upgrade with the provided `newScript`. However, the deployment of the new contract could be front-run'ed such that the states on the original contract cannot be migrated.

```
public static byte[] Upgrade(byte[] newScript, byte[] paramList, byte returnType,
    ContractPropertyState cps, string name, string version, string author, string email,
    string description)
{
    Assert(Runtime.CheckWitness(GetAdmin()), "upgrade: CheckWitness failed!");

    byte[] newContractHash = Hash160(newScript);

    Contract newContract = Contract.Migrate(newScript, paramList, returnType, cps, name,
        version, author, email, description);
    Runtime.Notify("upgrade", ExecutionEngine.ExecutingScriptHash, newContractHash);
    return newContractHash;
}
```

Listing 3.5: FlamingoSwapFactoryContract.Admin.cs

Specically, the `Upgrade()` function calls `Contract.Migrate()` to upgrade the contract. `Contract.Migrate()` migrates everything in the persistent storage of the current contract to the new contract when executed. In particular, the `Migrate()` method only transfers the contract storages when the target contract has not been deployed yet. Based on that, if the new contract has been deployed already, `Router` contract's states would not be migrated.

Recommendation Check whether the contract already exists before calling `Contract.Migrate()`.

Status The issue has been fixed by this commit: `04eb648`.

3.5 Violation of NEP-5 Standard in the Pair Contract

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FlamingoSwapPairContract.Nep5.cs`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [3]

Description

The NEP-5 proposal outlines a token standard for the Neo blockchain that provide a generalized interaction mechanism for tokenized Smart Contracts. Different from UTXO, the NEP5 assets are recorded in the contract storage area, and complete a transaction by updating account balance in the storage area. The `FlamingoSwapPairContract` contract implements the `Transfer()` function based on the NEP-5 standard. However, the current implementation has two minor issues, which violates the NEP-5 standard.

To be specific, when the `from` and `to` addresses are the same, the current implementation fails to fire the event but simply return `true`, as shown in line 82 of the code snippet below. In addition, according to the NEP-5 proposal, `Transfer()` should allow zero amount transfers with corresponding event emitted. But as shown in the code snippet, the assertion in line 73 rejects zero amount transfers by `Assert(amount > 0)`.

```
69 private static bool Transfer(byte[] from, byte[] to, BigInteger amount, byte[]
       callscript)
70 {
71     //Check parameters
72     Assert(from.Length == 20 && to.Length == 20, "The parameters from and to SHOULD be
           20-byte addresses.");
73     Assert(amount > 0, "The parameter amount MUST be greater than 0.");

75     if (!Runtime.CheckWitness(from) && from.AsBigInteger() != callscript.AsBigInteger())
76         return false;
77     StorageMap asset = Storage.CurrentContext.CreateMap(BalanceMapKey);
78     var fromAmount = asset.Get(from).AsBigInteger();
79     if (fromAmount < amount)
80         return false;
81     if (from == to)
82         return true;
83     ...
```

```
}
```

Listing 3.6: FlamingoSwapPairContract.Nep5.cs

Recommendation Fix the two violations as follows:

```
private static bool Transfer(byte[] from, byte[] to, BigInteger amount, byte[] callscript)
{
    //Check parameters
    Assert(from.Length == 20 && to.Length == 20, "The parameters from and to SHOULD be 20-byte addresses.");
    Assert(amount >= 0, "The parameter amount MUST be greater than 0.");

    if (!Runtime.CheckWitness(from) && from.AsBigInteger() != callscript.AsBigInteger())
        return false;
    StorageMap asset = Storage.CurrentContext.CreateMap(BalanceMapKey);
    var fromAmount = asset.Get(from).AsBigInteger();
    if (fromAmount < amount)
        return false;
    if (from == to) {
        Transferred(from, to, amount);
        return true;
    }
    ...
}
```

Listing 3.7: FlamingoSwapPairContract.Nep5.cs

Status The issue has been fixed by this commit: `04eb648`.

3.6 Missed Permission Checks in Mint(), Burn(), and Swap()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `FlamingoSwapPairContract.cs`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [3]

Description

Similar to Uniswap, Flamingo Swap allows liquidity providers (known as LPs for short) to add liquidity by transferring assets into the `Pair` contract through the `Router` contract. As a proof of providing liquidity, the liquidity tokens (LP tokens for short) are minted based on the amount of assets provided by an LP. When LPs want to withdraw the provided assets, the liquidity tokens would be burned based through the `Router` contract and `Pair` contract. Specifically, certain amount of liquidity tokens owned by the LP are sent to the `pairContract` contract with `SafeTransfer()` (line 155) followed by the `DynamicBurn()` call in line 157.

```
146 public static BigInteger[] RemoveLiquidity(byte[] sender, byte[] tokenA, byte[] tokenB,
        BigInteger liquidity, BigInteger amountAMin, BigInteger amountBMin, BigInteger
        deadLine)
147 {
148     //
149     Assert(Runtime.CheckWitness(sender), "Forbidden");
150     //
151     Assert((BigInteger) Runtime.Time <= deadLine, "Exceeded the deadline");


154     var pairContract = GetExchangePairWithAssert(tokenA, tokenB);
155     SafeTransfer(pairContract, sender, pairContract, liquidity);

157     var amounts = pairContract.DynamicBurn(sender);
158     var tokenAIsToken0 = tokenA.ToUInteger() < tokenB.ToUInteger();
159     var amountA = tokenAIsToken0 ? amounts[0] : amounts[1];
160     var amountB = tokenAIsToken0 ? amounts[1] : amounts[0];

162     Assert(amountA >= amountAMin, "INSUFFICIENT_A_AMOUNT");
163     Assert(amountB >= amountBMin, "INSUFFICIENT_B_AMOUNT");

165     return new BigInteger[] { amountA, amountB };
166 }
```

Listing 3.8: FlamingoSwapRouterContract.cs

Inside the `Burn()` function, the liquidity token balance of the `Pair` contract is retrieved in line 219. Later on, in line 227, all the liquidity tokens are burned. In addtion, the `Burn()` function is publicly available. It means whenever the `Pair` contract has liquidity tokens (e.g., someone accidentally sends in liquidity tokens), a bad actor could invoke the `Burn()` function and withdraw underlying assets.

```
212 public static object Burn(byte[] msgSender, byte[] toAddress)
213 {
214     var me = ExecutionEngine.ExecutingScriptHash;
215     var r = ReservePair;

217     var balance0 = DynamicBalanceOf(Token0, me);
218     var balance1 = DynamicBalanceOf(Token1, me);
219     var liquidity = BalanceOf(me);

221     //bool feeOn = MintFee(reserve0, reserve1);
222     var totalSupply = GetTotalSupply();
223     var amount0 = liquidity * balance0 / totalSupply;
224     var amount1 = liquidity * balance1 / totalSupply;

226     Assert(amount0 > 0 && amount1 > 0, "INSUFFICIENT_LIQUIDITY_BURNED");
227     BurnToken(me, liquidity);

229     SafeTransfer(Token0, me, toAddress, amount0);
```

```
    SafeTransfer(Token1, me, toAddress, amount1);
    ...
}
```

Listing 3.9: FlamingoSwapPairContract.cs

Besides, `Mint()` and `Swap()` also have the same issue. The business logic here is that the `Router` contract makes the caller of `RemoveLiquidity()` send her liquidity tokens (say 100 tokens) into the `Pair` contract and the caller gets underlying assets back with 100 LP tokens burned. Those 100 LP tokens should be the only LP tokens that the `Pair` has. Based on that, we suggest to add a whitelist mechanism to prevent other accounts from sending LP tokens into the `Pair` contract. This way, the `Pair` contract always gets LP tokens from the `Router` contract in the case of `RemoveLiquidity()`. No other case is possible.

Recommendation Whitelist the `Router` contract as the only `from` address while transferring LP tokens into the `Pair` contract.

Status The issue has been fixed by this commit: `90bdb87`.

3.7 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 Swap design and implementation. The Flamingo Swap is designed as an Auto Market Maker (AMM), providing liquidity to wrapped assets, FLM, and other NEP-5 based tokens. During the audit, we noticed that the current code base is well organized and those identified issues are promptly confirmed and fixed. As a precaution, again 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-703: Improper Check or Handling of Exceptional Conditions. https://cwe.mitre.org/data/definitions/703.html.

[2] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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