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 Proxy
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Edward Lo
Reviewed by	Jeff Liu, Chiachih Wu, Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Sep. 22, 2020	Xudong Shao	Final Release
1.0-rc2	Sep. 22, 2020	Xudong Shao	Release Candidate #2
1.0-rc1	Sep. 20, 2020	Xudong Shao	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 Proxy** 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 Proxy 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 Proxy

The Flamingo Proxy contract, also called NEP5Proxy, aims to faciliate contracts, deployed before cross-chain standard is ready, to perform cross-chain transactions. Users can lock any kind of assets into the Proxy contract with specific parameters, and the corresponding assets will be released to the designated account by unlock() method.

The basic information of Flamingo Proxy is as follows:

Table 1.1: Basic Information of Flamingo Proxy

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_crosschain/blob/master/NEP5Proxy/NEP5Proxy.cs (5cceba83)

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 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 Proxy 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	0	
High	1	■
Medium	1	■
Low	1	■
Informational	0	
Total	3	

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 1 high-severity vulnerability, 1 medium-severity vulnerability, and 1 low-severity vulnerability.

Table 2.1: Key Flamingo Proxy Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Front-Running in Init() Function	Time and State	Fixed
PVE-002	Low	Behavior Discrepancy in Lock() and Unlock() Functions	Coding Practices	Fixed
PVE-003	High	Lack of Sanity Check in Upgrade() Function	Time and State	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Front-Running in Init() Function

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `NEP5Proxy`
- Category: Time and State [3]
- CWE subcategory: CWE-362 [2]

Description

As shown in the following code snippet, there is no sanity check in the `Init()` of NEP5Proxy contract, which might lead to loss of the contract's ownership.

```
public static bool Init(byte[] _operator)
{
    assert(Storage.Get(OperatorKey).Length == 0, "init: operator exist: ".AsByteArray().
        Concat(Storage.Get(OperatorKey)).AsString());
    assert(Runtime.CheckWitness(_operator), "init: CheckWitness failed");
    Storage.Put(OperatorKey, _operator);
    InitEvent(_operator);
    return true;
}
```

Listing 3.1: NEP5Proxy.cs

The `Init()` function is used to set the `OperatorKey` in the contract after the deployment. However, anyone could front-run the `Init()` function to set the operator before the expected operator put in the storage.

Recommendation Make sure the `Init()` can only be called by a privilaged user, or combine the deployment and initialization of the contract in one transaction.

Status The issue has been fixed by this commit: `6fbdce9`

3.2 Behavior Discrepancy in the Lock() and Unlock() Functions

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: NEP5Proxy
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

There is a behavior discrepancy in the NEP5Proxy contract, which might accidentally lock a user's asset forever.

```
public static bool Lock(byte[] fromAssetHash, byte[] fromAddress, BigInteger toChainId,
    byte[] toAddress, BigInteger amount)
{
    assert(fromAssetHash.Length == 20, "lock: fromAssetHash SHOULD be 20-byte long.");
    assert(fromAddress.Length == 20, "lock: fromAddress SHOULD be 20-byte long.");
    assert(toAddress.Length > 0, "lock: toAddress SHOULD not be empty.");
    assert(amount > 0, "lock: amount SHOULD be greater than 0.");
    assert(!fromAddress.Equals(ExecutionEngine.ExecutingScriptHash), "lock: can not lock
        self");
    assert(!IsPaused(), "lock: proxy is locked");
    ...
}
```

Listing 3.2: NEP5Proxy.cs

```
public static bool Unlock(byte[] inputBytes, byte[] fromProxyContract, BigInteger
    fromChainId, byte[] callingScriptHash)
{
    //only allowed to be called by CCMC
    assert(callingScriptHash.Equals(CCMCScriptHash), "unlock: Only allowed to be called
        by CCMC.");

    byte[] proxyHash = Storage.Get(ProxyHashPrefix.Concat(fromChainId.ToByteArray()));

    // check the fromContract is stored, so we can trust it
    //assert(proxyHash.Equals(fromProxyContract), "unlock: fromProxyContract Not equal
        stored proxy hash.");
    if (fromProxyContract.AsBigInteger() != proxyHash.AsBigInteger())
    {
        Runtime.Notify("From proxy contract not found.");
        Runtime.Notify(fromProxyContract);
        Runtime.Notify(fromChainId);
        Runtime.Notify(proxyHash);
        return false;
    }
```

```
    assert(!IsPaused(), "lock: proxy is locked");

    // parse the args bytes constructed in source chain proxy contract, passed by multi-
        chain
    object[] results = DeserializeArgs(inputBytes);
    var toAssetHash = (byte[])results[0];
    var toAddress = (byte[])results[1];
    var amount = (BigInteger)results[2];
    assert(toAssetHash.Length == 20, "unlock: ToChain Asset script hash SHOULD be 20-
        byte long.");
    assert(toAddress.Length == 20, "unlock: ToChain Account address SHOULD be 20-byte
        long.");
    ...
}
```

Listing 3.3: NEP5Proxy.cs

User can lock his asset by `Lock()`, and get it back by `Unlock()`. These two functions all have corresponding sanity checks. However, `Lock()` only checks whether `toAddress` is empty, on the other hand, `Unlock()` makes sure `toAddress` is a valid address, namely, a 20-byte long address. This might lead to user's assets being locked forever if there is a typo.

Recommendation Align the sanity check in `Lock` and `Unlock` to check whether the address is a valid 20-byte long address.

Status The issue has been fixed by this commit: `8522b6c`

3.3 Lack of Sanity Check in Upgrade() Function

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `NEP5Proxy`
- Category: Time and State [3]
- CWE subcategory: CWE-362 [2]

Description

There is a lack of sanity check in the `Upgrade()` method of NEP5Proxy contract which might lead to unwanted behaviour.

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

```
    byte[] newContractHash = Hash160(newScript);


    assert(transferAssetsToNewContract(newContractHash), "upgrade: transfer asset into
        new contract hash failed!");

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

    Runtime.Notify(new object[] { "upgrade", ExecutionEngine.ExecutingScriptHash,
        newContractHash });
    return true;
}
```

Listing 3.4: NEP5Proxy.cs

The contract first transfers the tokens to the new contract, then it 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. For `Migrate()` method, it will only transfer the contract storages when the target contract has not been deployed yet.

Specifically, one can frontrun the deployment of the new contract so the migration won't transfer the storages to the new contract. Though what an attacker can do still depends on the new contract, this might not be the operator's expectation.

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: `460c9a7`

3.4 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 Proxy design and implementation. The contract is designed to facilitate cross-chain transactions. 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.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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.