SMART CONTRACT AUDIT REPORT

for

FLAMINGO

Prepared By: Shuxiao Wang

Hangzhou, China
September 20, 2020

Document Properties

Client	Flamingo
Title	Smart Contract Audit Report
Target	Proxy
Version	1.0-rc1
Author	Xudong Shao
Auditors	Xudong Shao, Edward Lo
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0-rc1	September 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 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 Proxy

Item	Description
Issuer	Flamingo
Website	https://www.oneswap.net/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 20, 2020

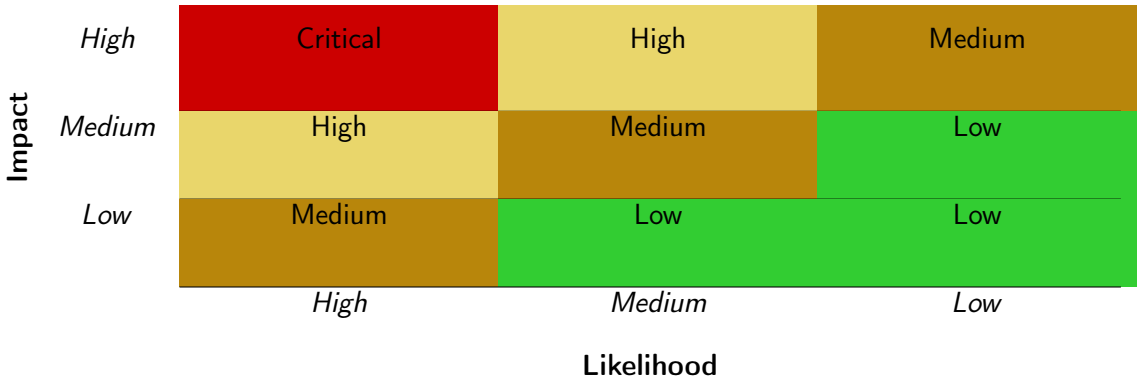
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/oneswap/oneswap_contract_ethereum (4194ac1)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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	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

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) [6], 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 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. We also measured the gas consumption of key operations with comparison with the popular `UniswapV2`. The purpose here is to not only statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool, but also understand the performance in a realistic setting.

The performance comparison shows that `OneSwap` outperforms `UniswapV2` in almost all aspects despite the additional support of limit orders in a DEX setting. In particular, the adoption of a proxy-based approach greatly reduces the gas cost for the creation of a new pair. Also, a variety of optimization efforts, including the clever use of `immutable` members, the packed design of orders and other data structures, as well as the efficient communication between proxy and logic, eventually pay off even with the burden of integrated limit order support in `OneSwap`. Among all audited DeFi projects, `OneSwap` is exceptional and really stands out in their extreme quest and dedication to maximize gas optimization.

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

Beside the performance measurement, 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 Proxy Audit Findings

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

Please refer to Section 3 for details.

3 Detailed Results

3.1 Possible Front-Running of Init()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `NEP5Proxy`
- Category: Time and State [4]
- CWE subcategory: CWE-362 [3]

Description

There is a lack of sanity check in the `Init()` of NEP5Proxy contract which might lead to loss of the contract's ownership.

```
77 public static bool Init(byte[] _operator)
78 {
79     assert(Storage.Get(OperatorKey).Length == 0, "init: operator exist: ".AsByteArray().
           Concat(Storage.Get(OperatorKey)).AsString());
80     assert(Runtime.CheckWitness(_operator), "init: CheckWitness failed");
81     Storage.Put(OperatorKey, _operator);
82     InitEvent(_operator);
83     return true;
84 }
```

Listing 3.1: NEP5Proxy.cs

The `Init()` function is used to set the `OperatorKey` in the contract after the deployment. However, everyone 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 users with permission, or, combine the deployment and initialization of the contract in one transaction.

Status The issue has been fixed by this commit: `6fbdce95b6322939ffae84a77276141bd637788f`

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

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: NEP5Proxy
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [2]

Description

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

```
217 public static bool Lock(byte[] fromAssetHash, byte[] fromAddress, BigInteger toChainId,
        byte[] toAddress, BigInteger amount)
218 {
219     assert(fromAssetHash.Length == 20, "lock: fromAssetHash SHOULD be 20-byte long.");
220     assert(fromAddress.Length == 20, "lock: fromAddress SHOULD be 20-byte long.");
221     assert(toAddress.Length > 0, "lock: toAddress SHOULD not be empty.");
222     assert(amount > 0, "lock: amount SHOULD be greater than 0.");
223     assert(!fromAddress.Equals(ExecutionEngine.ExecutingScriptHash), "lock: can not lock
            self");
224     assert(!IsPaused(), "lock: proxy is locked");
225     ...
226 }
```

Listing 3.2: NEP5Proxy.cs

```
250 public static bool Unlock(byte[] inputBytes, byte[] fromProxyContract, BigInteger
        fromChainId, byte[] callingScriptHash)
251 {
252     //only allowed to be called by CCMC
253     assert(callingScriptHash.Equals(CCMCScriptHash), "unlock: Only allowed to be called
            by CCMC.");

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

257     // check the fromContract is stored, so we can trust it
258     //assert(proxyHash.Equals(fromProxyContract), "unlock: fromProxyContract Not equal
            stored proxy hash.");
259     if (fromProxyContract.AsBigInteger() != proxyHash.AsBigInteger())
260     {
261         Runtime.Notify("From proxy contract not found.");
262         Runtime.Notify(fromProxyContract);
263         Runtime.Notify(fromChainId);
264         Runtime.Notify(proxyHash);
265         return false;
266     }
```

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

270     // parse the args bytes constructed in source chain proxy contract, passed by multi-
            chain
271     object[] results = DeserializeArgs(inputBytes);
272     var toAssetHash = (byte[])results[0];
273     var toAddress = (byte[])results[1];
274     var amount = (BigInteger)results[2];
275     assert(toAssetHash.Length == 20, "unlock: ToChain Asset script hash SHOULD be 20-
            byte long.");
276     assert(toAddress.Length == 20, "unlock: ToChain Account address SHOULD be 20-byte
            long.");
277     ...
278 }
```

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, `Unlcok` 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: `8522b6c730c2c1e7ede7482c1cafc3b5cdcae302`

3.3 Lack of Sanity Check in Upgrade()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `NEP5Proxy`
- Category: Time and State [4]
- CWE subcategory: CWE-362 [3]

Description

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

```
290 public static bool Upgrade(byte[] newScript, byte[] paramList, byte returnType,
        ContractPropertyState cps, string name, string version, string author, string email,
         string description)
291 {
292     assert(Runtime.CheckWitness(Storage.Get(OperatorKey)), "upgrade: CheckWitness failed
            !");
```

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


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

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

301     Runtime.Notify(new object[] { "upgrade", ExecutionEngine.ExecutingScriptHash,
            newContractHash });
302     return true;
303 }
```

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: `460c9a7bb8a2b1442cabaab1a9de8888a6058842`

3.4 Other Suggestions

Proxy merges the DEX support of traditional order book and automated market making. While it greatly pushes forward the DEX frontline, it also naturally inherits from well-known front-running or back-running issues plagued with current DEXs. For example, a large trade may be sandwiched by preceding addition into liquidity pool (via `mint()`) and tailgating removal of the same amount of liquidity (via `burn()`). Such sandwiching unfortunately causes a loss to other liquidity providers. Also, a large burn of the protocol token (via the built-in `buyback` mechanism) could be similarly sandwiched by preceding buys for increased token values. Similarly, a market order could be intentionally traded for a higher price if a malicious actor intentionally increases it by trading an earlier competing order. However, we need to acknowledge that these are largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Next, because the Solidity language is still maturing and it is common for new compiler versions to include changes that might bring unexpected compatibility or inter-version consistencies, it is always

suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.6;` instead of `pragma solidity >=0.6.6 or ^0.6.6;`.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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 noticed 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [8, 9, 10, 11, 13].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [14] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

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

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

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

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

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

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

[8] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[9] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[10] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[11] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[13] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[14] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.