SMART CONTRACT AUDIT REPORT

for

ALPHA FINANCE LAB

PeckShield

Prepared By: Shuxiao Wang

Hangzhou, China
Sep. 26, 2020

Document Properties

Client	Alpha Finance Lab
Title	Smart Contract Audit Report
Target	ALPHA Distribution
Version	1.0
Author	Chiachih Wu
Auditors	Chiachih Wu, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Sep. 26, 2020	Chiachih Wu	Final Release Version
1.0-rc1	Sep. 25, 2020	Chiachih Wu	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 **ALPHA Distribution** smart contract, 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 version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About ALPHA Distribution Smart Contract

Alpha Lending is a fully permissionless and decentralized lending protocol with algorithmic, autonomous interest rate. Built on Binance Smart Chain, Alpha Lending will support cross-chain assets and maximize alpha for lenders and borrowers. In Alpha Lending, the `distributor` is set to the `AlphaDistributor` contract, which distributes ALPHA to receivers such as the `ALPHASTAKE` contract. Users could `stake()` assets to such a receiver to get ALPHA rewards based on the amount and time of staking.

The basic information of ALPHA Distribution is as follows:

Table 1.1: Basic Information of ALPHA Distribution

Item	Description
Issuer	Alpha Finance Lab
Website	https://alphafinance.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Sep. 26, 2020

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

- https://github.com/AlphaFinanceLab/alpha-lending-smart-contract (33790b9)

1.2 About PeckShield

PeckShield Inc. [13] 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

PeckShield

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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

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

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) [7], 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 ALPHA Distribution 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, but also understand the performance in a realistic setting.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	0	
Informational	5	■■■■■
Total	5	

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 5 informational recommendations. Please refer to Section 3 for details.

Table 2.1: Key Audit Findings of ALPHA Distribution Protocol

ID	Severity	Title	Category	Status
PVE-001	Info.	Excessive Contract Calls in AlphaDistributor::poke()	Coding Practices	Confirmed
PVE-002	Info.	Zero Amount Transfers in VestingAlpha::claim()	Coding Practices	Fixed
PVE-003	Info.	Gas Optimization by Replacing Linked-List with Array	Coding Practices	Fixed
PVE-004	Info.	Privileged Interface to Withdraw ALPHA from AlphaDistributor	Business Logics	Confirmed
PVE-005	Info.	Optimized AlphaReleaseRule::getReleaseAmount()	Coding Practices	Fixed

PeckShield

3 Detailed Results

3.1 Excessive Contract Calls in AlphaDistributor::poke()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AlphaDistributor.sol`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1050 [2]

Description

In `AlphaDistributor` contract, the `poke()` function helps arbitrary users (or contracts) to trigger the ALPHA token distribution. While reviewing the `poke()` implementation, we noticed that there're excessive contract calls which could be optimized. Specifically, the `approve()` call in line 73 is followed by the `receiveAlpha()` call in line 74.

```
62  function poke() public override {
63    if (lastPokeBlock == block.number) {
64      return;
65    }
66    (IAlphaReceiver[] memory receivers, uint256[] memory values) = ruleSelector
67      .getAlphaReleaseRules(lastPokeBlock, block.number);
68    lastPokeBlock = block.number;
69    require(receivers.length == values.length, "Bad release rule length");
70    for (uint256 idx = 0; idx < receivers.length; ++idx) {
71      IAlphaReceiver receiver = receivers[idx];
72      uint256 value = values[idx];
73      alphaToken.approve(address(receiver), value);
74      receiver.receiveAlpha(value);
75    }
76  }
```

Listing 3.1: AlphaDistributor.sol

As we checked the example receiver, `AlphaStakePool` contract, the `receiveAlpha()` function has only one `transferFrom()` call. Since there's no other logic in the `receiveAlpha()` function, the two contract calls could be consolidated into one `alphaToken.transfer()`.

```
function receiveAlpha(uint256 _amount) external override {
  alphaToken.transferFrom(msg.sender, address(this), _amount);
}
```

Listing 3.2: AlphaStakePool.sol

Recommendation Combine the `approve()` and `transferFrom()` calls into one `transfer()` call.

Status As we discussed with the team, the `poke()` implementation should be kept as is for maintaining the receivers as generic as possible due to the fact that some business logic could be added into the receivers.

3.2 Zero Amount Transfers in VestingAlpha::claim()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `VestingAlpha.sol`
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

In `VestingAlpha` contract, the `claim()` function allows users to claim the vested ALPHA tokens. While reviewing the implementation, we identified that some cases may lead to zero amount transfers with `ReceiptClaimed()` events emitted, which is not necessary.

```
function claim(uint256 _receiptID) external override {
  require(_receiptID < receipts.length, "Receipt ID not found");
  Receipt storage receipt = receipts[_receiptID];
  require(msg.sender == receipt.recipient, "Only receipt recipient can claim this
      receipt");
  uint256 duration = now.sub(receipt.createdAt) < vestingDuration
    ? now.sub(receipt.createdAt)
    : vestingDuration;
  uint256 pending = duration.mul(receipt.amount).div(vestingDuration).sub(receipt.
      claimedAmount);
  receipt.claimedAmount = receipt.claimedAmount.add(pending);
  alphaToken.transfer(receipt.recipient, pending);
  emit ReceiptClaimed(_receiptID, pending);
}
```

Listing 3.3: VestingAlpha.sol

Specifically, `claim()` computes the `pending` which is part of the `receipt.amount` based on the time since the `receipt` is created. The `receipt.claimedAmount` keeps the already claimed amount (i.e., `pending`). Based on that, when `pending == 0`, lines 100-102 could be skipped. In addition, when `receipt.claimedAmount` reaches `receipt.amount`, the rest of the function could be skipped. Therefore, as we confirmed that the `msg.sender` can claim the `receipt` in line 95, `claim()` could revert or return directly when `receipt.claimedAmount == receipt.amount`.

Recommendation Add `receipt.claimedAmount < receipt.amount` and `pending > 0` sanity checks into `claim()`.

```
92   function claim(uint256 _receiptID) external override {
93     require(_receiptID < receipts.length, "Receipt ID not found");
94     Receipt storage receipt = receipts[_receiptID];
95     require(msg.sender == receipt.recipient, "Only receipt recipient can claim this
         receipt");
96     require(receipt.claimedAmount < receipt.amount, "Nothing to claim");
97     uint256 duration = now.sub(receipt.createdAt) < vestingDuration
98       ? now.sub(receipt.createdAt)
99       : vestingDuration;
100    uint256 pending = duration.mul(receipt.amount).div(vestingDuration).sub(receipt.
         claimedAmount);
101    if ( pending > 0 ) {
102      receipt.claimedAmount = receipt.claimedAmount.add(pending);
103      alphaToken.transfer(receipt.recipient, pending);
104      emit ReceiptClaimed(_receiptID, pending);
105    }
106  }
```

Listing 3.4: VestingAlpha.sol

Status This issue had been addressed in this commit: a6b25bc

3.3 Gas Optimization by Replacing Linked-List with Array

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AlphaReleaseRuleSelector.sol`
- Category:
- CWE subcategory:

Description

In `AlphaReleaseRuleSelector` contract, we noticed that the `receiverList` linked-list is implemented to maintian the list of `receivers`. As a character of linked-list, when the owner needs to `setAlphaReleaseRule()` the `_rule` of a specific `_receiver`, a new entry is added as the first entry of the `receiverList` linked-list if `_receiver` is not added before (line 57-61).

```
function setAlphaReleaseRule(IAlphaReceiver _receiver, IAlphaReleaseRule _rule)
  external
  onlyOwner
{
  // Add new rules
  if (receiverList[address(_receiver)] == address(0)) {
    receiverList[address(_receiver)] = receiverList[HEAD];
    receiverList[HEAD] = address(_receiver);
    ruleCount++;
  }
  // Set the release rule to the receiver
  rules[address(_receiver)] = _rule;
  emit AlphaReleaseRuleUpdated(address(_receiver), address(_rule));
}
```

Listing 3.5: AlphaReleaseRuleSelector.sol

When the owner needs to retrieve all receivers and the corresponding amounts of ALPHA to be released, the linked-list needs to be traversed in `getAlphaReleaseRules()` function.

```
function getAlphaReleaseRules(uint256 _fromBlock, uint256 _toBlock)
  external
  override
  view
  returns (IAlphaReceiver[] memory, uint256[] memory)
{
  IAlphaReceiver[] memory receivers = new IAlphaReceiver[](ruleCount);
  uint256[] memory amounts = new uint256[](ruleCount);
  address currentReceivers = receiverList[HEAD];
  for (uint256 i = 0; i < ruleCount; i++) {
    receivers[i] = IAlphaReceiver(currentReceivers);
    IAlphaReleaseRule releaseRule = rules[currentReceivers];
    amounts[i] = releaseRule.getReleaseAmount(_fromBlock, _toBlock);
    currentReceivers = receiverList[currentReceivers];
  }
  return (receivers, amounts);
}
```

Listing 3.6: AlphaReleaseRuleSelector.sol

Due to the fact that the new `_receiver` is added at the beginning of the linked-list, we believe the linked-list could be replaced with an array. The new entry could be simply `push()` into the array in the `setAlphaReleaseRule()` function. Furthermore, the `getAlphaReleaseRules()` function could be refactored with an array to reduce gas consumption.

Recommendation Replace the linked-list implementation to array.

Status This issue had been addressed in this commit: 7a8517f

3.4 Privileged Interface to Withdraw ALPHA from AlphaDistributor

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AlphaDistributor.sol`
- Category:
- CWE subcategory:

Description

In `AlphaDistributor` contract, the `withdrawAlpha()` function allows the owner to withdraw `_amount` of ALPHA tokens. Since `AlphaDistributor` is source of all ALPHA distribution, users could trigger the distribution by calling the `poke()` function. However, if the balance of ALPHA is not enough, the `poke()` call would be reverted. Based on that, the privileged interface allow the owner to disable the ALPHA distribution indirectly.

```
function withdrawAlpha(uint256 _amount) external onlyOwner {
  alphaToken.transfer(msg.sender, _amount);
  emit WithdrawAlpha(msg.sender, _amount);
}
```

Listing 3.7: AlphaDistributor.sol

Recommendation Remove the privileged interface or set the owner as a multi-sig/timelock contract.

Status As we discussed with the team, the `withdrawAlpha()` function is used for migration. We suggest to deploy a multi-sig contract or a timelock contract as the `owner` to avoid privileged interface from being misused.

3.5 Optimized AlphaReleaseRule::getReleaseAmount()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AlphaReleaseRule.sol`
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [3]

Description

In `AlphaReleaseRule` contract, the `getReleaseAmount()` function allows the caller to get the amount of ALPHA to be released in the period of `_fromBlock` to `_toBlock`. While reviewing the implementation, we came up with an optimization idea which reduces around 20 gas in each iteration of the while-loop.

```
51 function getReleaseAmount(uint256 _fromBlock, uint256 _toBlock)
52   external
53   override
54   view
55   returns (uint256)
56 {
57   uint256 lastBlock = startBlock.add(tokensPerBlock.length.mul(blockPerWeek));
58   if (_toBlock <= startBlock || lastBlock <= _fromBlock) {
59     return 0;
60   }
61   uint256 fromBlock = _fromBlock > startBlock ? _fromBlock : startBlock;
62   uint256 toBlock = _toBlock < lastBlock ? _toBlock : lastBlock;
63   uint256 week = findWeekByBlockNumber(fromBlock);
64   uint256 totalAmount = 0;
65   while (fromBlock < toBlock) {
66     uint256 lastBlockInWeek = findLastBlockOnThisWeek(fromBlock);
67     lastBlockInWeek = toBlock < lastBlockInWeek ? toBlock : lastBlockInWeek;
68     totalAmount = totalAmount.add(lastBlockInWeek.sub(fromBlock).mul(tokensPerBlock[week
         ]));
69     week = week.add(1);
70     fromBlock = lastBlockInWeek;
71   }
72   return totalAmount;
73 }
```

Listing 3.8: AlphaReleaseRule.sol

Specifically, the `lastBlockInWeek` is set to the first block in `week+1` in the first line of the while-loop (line 66) where `week` is derived from `fromBlock` (line 63). Based on that, in the second iteration of the while-loop, `lastBlockInWeek` is set to the first block in `week+2`, which equals the previous `lastBlockInWeek` + `blockPerWeek`.

```
function findLastBlockOnThisWeek(uint256 _block) public view returns (uint256) {
  require(_block >= startBlock, "the block number must more than or equal start block");
  return
    _block.sub(startBlock).div(blockPerWeek).mul(blockPerWeek).add(blockPerWeek).add(
        startBlock);
}
```

Listing 3.9: AlphaReleaseRule.sol

As shown in the code snippet above, the `findLastBlockOnThisWeek()` function has 2 `add()`, 1 `sub()`, 1 `div()`, and 1 `mul()` operations. If we could replace a `findLastBlockOnThisWeek()` call with the `add()` operation mentioned earlier, we could reduce around 20 gas cost. When `fromBlock` is far from `toBlock`, the gas optimization would be significant. Besides, the case `fromBlock >= toBlock` should be filtered out in the beginning of the function.

Recommendation Refactor `getReleaseAmount()` as follows:

```
function getReleaseAmount(uint256 _fromBlock, uint256 _toBlock)
  external
  override
  view
  returns (uint256)
{
  uint256 lastBlock = startBlock.add(tokensPerBlock.length.mul(blockPerWeek));
  if (fromBlock >= toBlock || _toBlock <= startBlock || lastBlock <= _fromBlock) {
    return 0;
  }
  uint256 fromBlock = _fromBlock > startBlock ? _fromBlock : startBlock;
  uint256 toBlock = _toBlock < lastBlock ? _toBlock : lastBlock;
  uint256 week = findWeekByBlockNumber(fromBlock);
  uint256 totalAmount = 0;
  uint256 lastBlockInWeek = findLastBlockOnThisWeek(fromBlock);
  while (fromBlock < toBlock) {
    lastBlockInWeek = toBlock < lastBlockInWeek ? toBlock : lastBlockInWeek;
    totalAmount = totalAmount.add(lastBlockInWeek.sub(fromBlock).mul(tokensPerBlock[week
        ]));
    week = week.add(1);
    fromBlock = lastBlockInWeek;
    lastBlockInWeek = lastBlockInWeek.add(blockPerWeek);
  }
  return totalAmount;
}
```

Listing 3.10: AlphaReleaseRule.sol

Status This issue had been addressed in this commit: fb22bb4

3.6 Other Suggestions

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 design and implementation of the ALPHA Distribution smart contract. The system presents a clean and consistent design that makes it distinctive and valuable alternative to current yield farming offerings. 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

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 [9, 10, 11, 12, 14].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [15] 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-1050: Excessive Platform Resource Consumption within a Loop. https://cwe.mitre.org/data/definitions/1050.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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

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

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

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

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

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

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

PeckShield