SMART CONTRACT AUDIT REPORT

for

YAM FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China

August 19, 2020

Document Properties

Client	Yam Finance
Title	Smart Contract Audit Report
Target	YAMv2
Version	1.0
Author	Xuxian Jiang
Auditors	Chiachih Wu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	August 19, 2020	Xuxian Jiang	Final Release

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 **YAMv2** 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 version of smart contracts is well-designed and the implementation is of high-quality and in accordance with the purpose despite the presence of several (minor) issues. This document outlines our audit results.

1.1 About YAMv2

Migrated from YAMv1, YAMv2 has two main components: `YAMv2 ERC-20` token and `YAMv2Migration`. The first component is in essence a standard ERC-20 token with no rebases to serve as placeholder and the second component performs the actual migration that allows all YAMv1 holders to burn their YAMv1 tokens in order to mint their share of YAMv2 tokens. Note that the number of YAMv2 tokens received after migration will be based upon each holder's YAMv1 `yamBalances`, which remains constant regardless of rebases.

The basic information of YAMv2 is as follows:

Table 1.1: Basic Information of YAMv2

Item	Description
Issuer	Yam Finance
Website	https://yam.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 19, 2020

In the following, we show the repository of reviewed code and the commit hash value used in this

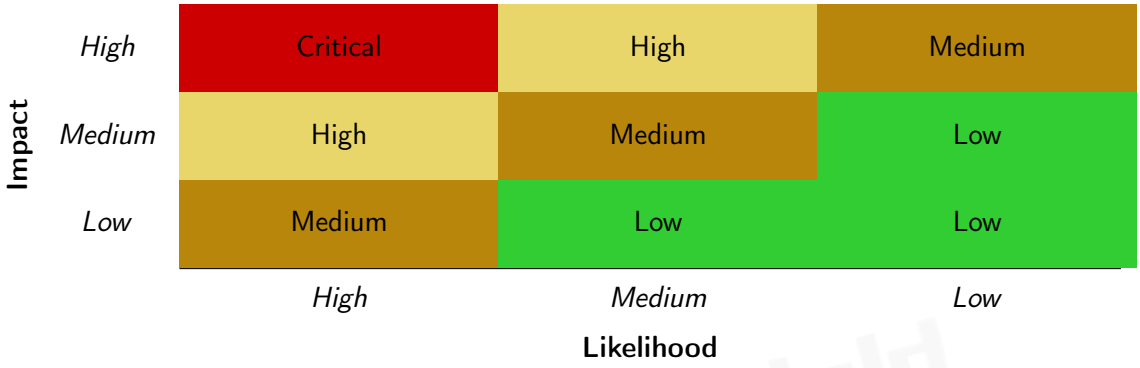
audit:

- https://github.com/yam-finance/yam-migration.git (5a79aec)

1.2 About PeckShield

PeckShield Inc. [15] 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 [10]:

- 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) [9], 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 an 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 YAMv2 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 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.

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

We have so far identified a list of potential issues. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined three of them need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key YAMv2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Unused Library Removal	Coding Practices	Fixed
PVE-002	Low	No Migration Before Token Initialization	Security Features	Fixed
PVE-003	Low	(Minor) Non-Compliance of ERC-20 Standard	Coding Practices	Confirmed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Unused Library Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: YAMv2
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [6]

Description

YAMv2 makes good use of a few library functions, such as Context, SafeERC20, IERC20, SafeMath, Address, and Ownable, to facilitate its various calculations and computations. For example, the YAMv2 smart contract imports at least four libraries, i.e., Context, IERC20, SafeMath and Address.

If we examine the Address library, it provides two handy routines: isContract() and sendValue(). The first one determines whether the provided account is a contract or not while the second one provides a replacement of Solidity's transfer() by forwarding all available gas and reverting on errors (due to introduced gas cost change of certain opcodes in EIP1884 [2]). However, both these two routines are not used anymore in current code base. Therefore, the Address library can be safely removed.

```
3  pragma solidity ^0.6.0;
4
5  import "./lib/Context.sol";
6  import "./lib/IERC20.sol";
7  import "./lib/SafeMath.sol";
8  import "./lib/Address.sol";
9
10
11
12 /**
13  * @title YAMv2 Token
14  * @dev YAMv2 Mintable Token with migration from legacy contract. Used to signal
15  *      for protocol changes in v3.
```

```
16   */
17 contract YAMv2 is Context, IERC20 {
18     using SafeMath for uint256;
19     using Address for address;
20
21     ...
22
23 }
```

Listing 3.1: YAMv2.sol

Recommendation Remove the `Address` library in current code base and delete their reference in `YAMv2` (line 8) as well.

Result This issue has been addressed by removing the unused `Address` library and reflected in this commit: 509db8e8b82edc0bbaf2e68fd7c35acf44fc9ec1.

3.2 No Migration Before Token Initialization

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: N/A
- Target: `YAMv2Migration.sol`
- Category: Security Features [7]
- CWE subcategory: CWE-284 [4]

Description

The migration logic proceeds by first determining the calling user's balance in `YAMv1`, next burning the balance (by essentially transferring the balance to an address no one has access to the corresponding private key), and then minting corresponding `YAMv2` amount to the user. To properly kick-off the migration process, the smart contract ensures that the migration process takes place only during the specified migration period: `[startTime, startTime + migrationDuration)`. Both `startTime` and `migrationDuration` are constants hard-coded in the contact implementation.

Our analysis shows that there is one more requirement that needs to be satisfied before actually starting the migration process, i.e., the `YAMv2` token address has been properly initialized. A non-initialized `YAMv2` address (default `address(0)`) could result in the attempt to burn `YAMv1` tokens. Fortunately, the `address(0).mint()` call would revert the transaction and hence no loss will be caused.

```
55     /**
56      * @dev Migrate a users' entire balance
57      *
58      * One way function. YAMv1 tokens are BURNED. YAMv2 tokens are minted.
59      */
60     function migrate() public virtual {
```

```
61          require(block.timestamp >= startTime, "!started");
62          require(block.timestamp < startTime + migrationDuration, "migration ended");
63
64          // current scalingFactor
65          uint256 scalingFactor = YAM(yam).yamsScalingFactor();
66
67          // gets the yamValue for a user.
68          uint256 yamValue = YAM(yam).balanceOfUnderlying(_msgSender());
69
70          // since balanceOfUnderlying has more decimals than balanceOf,
71          // we cant transfer the entirety of balanceOfUnderlying.
72          // we have no method of decrementing balanceOfUnderlying directly,
73          // as this is not intended use.
74          // remainder is guaranteed to be less than 10^-18
75          // not a perfect solution, but is not profitable to do in almost any
76          // gas environment
77          // requires migrating 1000000000000000000 times to mint an additional
78          // underlying yam
79          require(YAM(yam).balanceOf(_msgSender()) > 0, "No yams");
80
81          // gets transferFrom amount by multiplying by scaling factor / 10**24
82          // equivalent to balanceOf, but we need underlyingAmount later
83          uint256 transferAmount = yamValue.mul(scalingFactor).div(internalDecimals);
84
85          // BURN YAM - UNRECOVERABLE.
86          SafeERC20.safeTransferFrom(
87              IERC20(yam),
88              _msgSender(),
89              address(0x000000000000000000000000000000000000dEaD),
90              transferAmount
91          );
92
93          // mint new YAMv2, using yamValue (1e24 decimal token, to match internalDecimals
                )
94          YAMv2(yamV2).mint(_msgSender(), yamValue);
95      }
```

Listing 3.2: YAMv2Migration.sol

For elaboration, we show the `migrate()` routine above. Our analysis shows that the burned amount, i.e., `transferAmount`, can be computed in two different ways:

1. `transferAmount = yamValue.mul(scalingFactor).div(internalDecimals)` (line 83)
2. `transferAmount = YAM(yam).balanceOf(_msgSender())` (line 79).

The above two approaches lead to the same result and we can simply take one with less gas cost. As a result, the above logic can be slightly simplified by removing `require(YAM(yam).balanceOf(_msgSender())> 0, "No yams")` (line 79) and adding `require(transferAmount > 0, "No yams")` (right after line 83).

Recommendation Ensure that the migration process does not take place until the `YAMv2` token address has been properly initialized (in addition to being within the specified 3-day time window of migration). And we can further (slightly) simplify the logic as follows:

```
55      /**
56       * @dev Migrate a users' entire balance
57       *
58       * One way function. YAMv1 tokens are BURNED. YAMv2 tokens are minted.
59       */
60      function migrate() public virtual {
61          require(token_initialized, "YAMv2 not set yet");
62          require(block.timestamp >= startTime, "!started");
63          require(block.timestamp < startTime + migrationDuration, "migration ended");
64
65          // current scalingFactor
66          uint256 scalingFactor = YAM(yam).yamsScalingFactor();
67
68          // gets the yamValue for a user.
69          uint256 yamValue = YAM(yam).balanceOfUnderlying(_msgSender());
70
71          // gets transferFrom amount by multiplying by scaling factor / 10**24
72          // equivalent to balanceOf, but we need underlyingAmount later
73          uint256 transferAmount = yamValue.mul(scalingFactor).div(internalDecimals);
74          require(transferAmount > 0, "No yams");
75
76
77          // BURN YAM - UNRECOVERABLE.
78          SafeERC20.safeTransferFrom(
79              IERC20(yam),
80              _msgSender(),
81              address(0x000000000000000000000000000000000000dEaD),
82              transferAmount
83          );
84
85          // mint new YAMv2, using yamValue (1e24 decimal token, to match internalDecimals
                )
86          YAMv2(yamV2).mint(_msgSender(), yamValue);
87      }
```

Listing 3.3: YAMv2Migration.sol (revised)

Result This issue has been addressed by adding the suggested requirement and fixed in this commit: e41a9fc0be4fb994ab36c20a5793e26b21aafd33.

3.3 (Minor) Non-Compliance of ERC-20 Standard

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: YAMv2
- Category: Coding Practices [8]
- CWE subcategory: CWE-547 [5]

Description

ERC-20 is a community-driven standard that specifies basic functionality to transfer tokens, as well as allow tokens to be approved so they can be later spent by another on-chain third party. The defined interfaces include widely used `transfer()`, `transferFrom()`, `approve()` etc.

We note the `transfer()` implementation slightly deviates from the ERC-20 standard. Specifically, the standard explicitly makes it clear that *"Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event"* [3]. But there is no restriction imposed on the recipient!

The current implementation, however, shows that `transfer()` to the recipient `address(0)` is explicitly prohibited (see the related code snippet below - line 201). This is an additional requirement beyond the standard.

```
185     /**
186      * @dev Moves tokens `amount` from `sender` to `recipient`.
187      *
188      * This is internal function is equivalent to {transfer}, and can be used to
189      * e.g. implement automatic token fees, slashing mechanisms, etc.
190      *
191      * Emits a {Transfer} event.
192      *
193      * Requirements:
194      *
195      * - `sender` cannot be the zero address.
196      * - `recipient` cannot be the zero address.
197      * - `sender` must have a balance of at least `amount`.
198      */
199     function _transfer(address sender, address recipient, uint256 amount) internal
         virtual {
200         require(sender != address(0), "ERC20: transfer from the zero address");
201         require(recipient != address(0), "ERC20: transfer to the zero address");
202
203         _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount
             exceeds balance");
204         _balances[recipient] = _balances[recipient].add(amount);
205         emit Transfer(sender, recipient, amount);
206     }
```

Listing 3.4: YAMv2.sol

In the meantime, we notice that it is also a common practice in blocking the token transfers to `address(0)` as it prevents certain insufficiently tested software (that may accidently provide an empty argument as recipient) from sending the funds into the void.

Recommendation This is really optional and the above ERC-20 deviation is considered minor. Our suggestion is to be compliant with the ERC-20 standard if at all possible.

Result The `YAMv2` smart contract is heavily based on the default OpenZeppelin implementation of ERC-20 standard: `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol`. We both agree that we can just leave it as is.

3.4 Other Suggestions

Due to the fact that compiler upgrades 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 0.6.0;` instead of `pragma solidity >=0.6.0;`.

Moreover, 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 YAMv2 design and implementation. Overall, YAM presents a unique offering in current DeFi ecosystem and we are impressed by the design and implementation. 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.

Note: We received full payment from `Yam Finance` for our security audit service. In the following, we disclose the related transaction hash that completed the payment:

`https://etherscan.io/tx/0xf1dcb609401efd8e6ed4e74908f9fb301d4a052656f98736ec7cab0f30ed6787`.

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 [11, 12, 13, 14, 16].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [17] 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] The Ethereum Foundation. EIP-1884: Repricing for Trie-Size-Dependent Opcodes. https://eips.ethereum.org/EIPS/eip-1884.

[3] The Ethereum Foundation. EIP-20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[4] MITRE. CWE-287: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html.

[5] MITRE. CWE-547: Use of Hard-coded, Security-relevant Constants. https://cwe.mitre.org/data/definitions/547.html.

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

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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

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

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

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

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

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