SMART CONTRACT AUDIT REPORT

for

HARMONY

Prepared By: Shuxiao Wang

Hangzhou, China
Oct. 21, 2020

Document Properties

Client	Harmony
Title	Smart Contract Audit Report
Target	Ethhmy-bridge
Version	1.0
Author	Chiachih Wu
Auditors	Chiachih Wu, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Oct. 21, 2020	Chiachih Wu	Final Release
1.0-rc2	Aug. 28, 2020	Chiachih Wu	Release Candidate #2
1.0-rc1	Aug. 24, 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 source code of the **Ethhmy-bridge**, 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 Ethhmy-bridge

Ethhmy-bridge is a two-way bridge between Ethereum and Harmony blockchains, with trusted validators in between to facilitate and verify token transfers between these two blockchains. Users can transfer ERC20 tokens such as Binance USD (BUSD) from Ethereum to Harmony chain by locking the ERC20 tokens into the bridge validator account, then receive HRC20 version BUSD tokens to their account on Harmony chain, and later users can also request to burn the HRC20 BUSD tokens she received, which in turn would release the locked ERC20 BUSD tokens to her Ethereum account. And the bridge works for both directions, token transfers from Harmony to Ethereum can be faciliated in a similar way.

The basic information of Ethhmy-bridge is as follows:

Table 1.1: Basic Information of Ethhmy-bridge

Item	Description
Issuer	Harmony
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Oct. 21, 2020

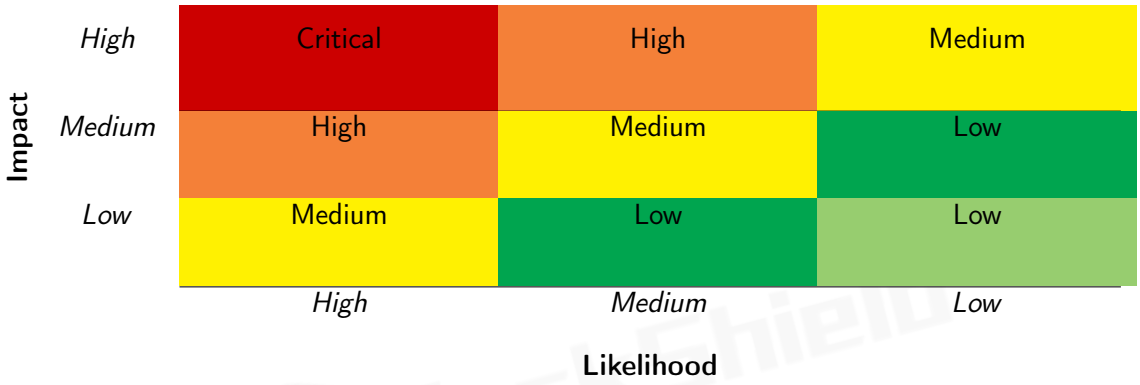
In the following, we show the repository of reviewed code used in this audit.

- https://github.com/harmony-one/ethhmy-bridge (a8b7000)
- https://github.com/harmony-one/ethhmy-bridge (85b7a90)
- https://github.com/harmony-one/ethhmy-bridge (490c0b2)

1.2 About PeckShield

PeckShield Inc. [10] 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 [5]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [4], 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 Ethhmy-bridge implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	0	
Low	2	■■
Informational	1	■
Total	5	

We have so far 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 2 high-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Ethhmy-bridge Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logics	Fixed
PVE-002	Low	Eternally Locked Tokens	Business Logics	Fixed
PVE-003	Info.	Potential Denial-of-Service Risks in deny()	Business Logics	Fixed
PVE-004	High	Extra Auth Privilege in mintToken()/unlockToken()	Business Logics	Fixed
PVE-005	High	Extra Auth Privilege in addToken()/removeToken()	Business Logics	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `EthManager.sol`
- Category: Business Logics [3]
- CWE subcategory: CWE-841 [2]

Description

In Ethhmy-bridge, the public function, `lockToken()`, allows users to transfer in ERC20 tokens and generate corresponding tokens on the Harmony blockchain. Specifically, the `Locked` event emitted within a `lockToken()` transaction indicates the amount and the recipient of a specific token to be minted at the Harmony side. However, when the `ethToken_` is a deflationary token (e.g., STA, STONK, or even USDT), the `amount` of token received in line 50 could be less than what we expected. Therefore, more tokens than the locked amount could be minted at the Harmony side. Later on, when someone burns tokens on the Harmony chain for unlocking ERC20 tokens, the `EthManager` contract may not have enough ERC20 balance to unlock.

```
49      function lockToken(uint256 amount, address recipient) public {
50          ethToken_.safeTransferFrom(msg.sender, address(this), amount);
51          emit Locked(address(ethToken_), msg.sender, amount, recipient);
52      }
```

Listing 3.1: EthManager.sol

Recommendation Check the balance before and after the `safeTransferFrom()` call to ensure the book-keeping amount is accurate. Another alternative solution is using non-deflationary tokens as the `ethToken_` but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Result This issue has been addressed by checking the token balance of `msg.sender` before and after the `safeTransferFrom()` call in this commit: 490c0b2.

3.2 Eternally Locked Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Proposal.sol`
- Category: Business Logics [3]
- CWE subcategory: CWE-841 [2]

Description

In Ethhmy-bridge, the public function, `lockToken()`, allows users to transfer in ERC20 tokens and generate corresponding tokens on the Harmony blockchain. Specifically, the `Locked` event emitted within a `lockToken()` transaction indicates the amount and the recipient of a specific token to be minted at the Harmony side. As a common practice (including the OpenZeppelin's implmentation), `mint()` to a zero address is prohibited in ERC20 contracts. Based on that, if the `recipient` is a zero address, there will be no token minted on the Harmony chain. Therefore, those tokens locked in the `EthManager` contract will be locked eternally, which is not the case we expected.

```
49      function lockToken(uint256 amount, address recipient) public {
50          ethToken_.safeTransferFrom(msg.sender, address(this), amount);
51          emit Locked(address(ethToken_), msg.sender, amount, recipient);
52      }
```

Listing 3.2: EthManager.sol

Recommendation Ensure the `recipient` is a non-zero address.

```
49      function lockToken(uint256 amount, address recipient) public {
50          require(recipient != address(0), "recipient is a zero address");
51          require(amount > 0, "zero token locked");
52          ethToken_.safeTransferFrom(msg.sender, address(this), amount);
53          emit Locked(address(ethToken_), msg.sender, amount, recipient);
54      }
```

Listing 3.3: EthManager.sol

In addition, locking zero token is not necessary to be sync-ed to Harmony chain. We could get rid of it by requiring `amount > 0`.

Result This issue has been addressed by checking the `recipient` in this commit: 490c0b2.

3.3 Potential Denial-of-Service Risks in deny()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: High
- Target: `Proposal.sol`
- Category: Business Logics [3]
- CWE subcategory: CWE-841 [2]

Description

In Ethhmy-bridge, an authorized user could remove a `guy` from the authorized list by the `deny()` function. However, an authorized user could `deny()` an arbitrary `guy` including herself, which means all authorized users could be removed from the list. Whenever the last authorized user is `deny()`'ed, no one can `unlockToken()` anymore.

```
26      function deny(address guy) external auth {
27          wards[guy] = 0;
28      }
```

Listing 3.4: EthManager.sol

Recommendation Ensure the `owner` would not be `deny()`'ed. Note that the `owner` needs to be stored in the storage inside the `constructor()`.

```
26      function deny(address guy) external auth {
27          require(guy != owner, "cannot deny the owner");
28          wards[guy] = 0;
29      }
```

Listing 3.5: EthManager.sol

Result This issue has been addressed by avoiding `deny()` the `owner` in this commit: 490c0b2.

3.4 Extra Auth Privilege in mintToken()/unlockToken()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `BUSDEthManager.sol`, `BUSDHmyManager.sol`, `LINKEthManager.sol`, `LINKHmyManager.sol`, `EthManager.sol`, `HmyManager.sol`
- Category: Business Logics [3]
- CWE subcategory: CWE-841 [2]

Description

In Ethhmy-bridge, an authorized user can add a `guy` from the authorized list by calling the `rely()` function, and the authorized list in EthManager contract is used to call `unlockToken()` to unlock tokens, as shown in the code snippet below.

```
22  mapping(address => uint256) public wards;

24  function rely(address guy) external auth {
25      wards[guy] = 1;
26  }
```

Listing 3.6: EthManager.sol

In the code below, the caller of the `unlockToken()` function must be `auth`, and if auth is an EOA and its account/server is compromised for some reason, the hacker could drain all the token assets by calling `unlockToken()`, therefore this privilege of auth is a huge risk for the system. Similar issues exist in other contracts such as BUSDEthManager, BUSDHmyManager, LINKEthManager, LINKHmyManager, EthManager, and HmyManager.

```
68  /**
69  * @dev unlock tokens after burning them on harmony chain
70  * @param amount amount of unlock tokens
71  * @param recipient recipient of the unlock tokens
72  * @param receiptId transaction hash of the burn event on harmony chain
73  */
74 function unlockToken(
75     uint256 amount,
76     address recipient,
77     bytes32 receiptId
78 ) public auth {
79     require(
80         !usedEvents_[receiptId],
81         "EthManager/The burn event cannot be reused"
82     );
83     usedEvents_[receiptId] = true;
84     ethToken_.safeTransfer(recipient, amount);
85     emit Unlocked(amount, recipient);
86 }
```

Listing 3.7: EthManager.sol

Recommendation To mitigate the risk, we recommend a multi-sig wallet contract maintained by multiple validator nodes.

Result This issue has been addressed by adding multi-sig wallet in this commit: a8b7000.

3.5 Extra Auth Privilege in addToken()/removeToken()

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `TokenManager.sol`
- Category: Business Logics [3]
- CWE subcategory: CWE-841 [2]

Description

In Ethhmy-bridge, the public functions, `addToken()/removeToken()`, allow the superuser to add/remove tokens. As shown in the following code, `addToken()` function allows `auth` to add tokens into the system. Similar to the issue we described in PVE-004, if auth is an EOA and its account/server is compromised for some reason, the hacker could add/remove tokens at will, therefore this privilege of auth is a huge risk for the system.

```
30      /**
31       * @dev map ethereum token to harmony token and emit mintAddress
32       * @param ethTokenAddr address of the ethereum token
33       * @return mintAddress of the mapped token
34       */
35      function addToken(
36          address ethTokenAddr,
37          string memory name,
38          string memory symbol,
39          uint8 decimals
40      ) public auth returns (address) {
41          require(
42              ethTokenAddr != address(0),
43              "TokenManager/ethToken is a zero address"
44          );
45          require(
46              mappedTokens[ethTokenAddr] == address(0),
47              "TokenManager/ethToken already mapped"
48          );

50          BridgedToken bridgedToken = new BridgedToken(
51              ethTokenAddr,
52              name,
53              symbol,
54              decimals
55          );
56          address bridgedTokenAddr = address(bridgedToken);

58          // store the mapping and created address
59          mappedTokens[ethTokenAddr] = bridgedTokenAddr;

61          // assign minter role to the caller
```

```
62          bridgedToken.addMinter(msg.sender);

64          emit TokenMapAck(ethTokenAddr, bridgedTokenAddr);
65          return bridgedTokenAddr;
66      }
```

Listing 3.8: TokenManager.sol

Recommendation To mitigate the risk, we recommend a multi-sig wallet contract maintained by multiple validator nodes.

Result This issue has been addressed by adding multi-sig wallet in this commit: a8b7000.

3.6 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 0.5.0;` instead of `pragma solidity >=0.5.0;`.

In addition, there is a known compiler issue that in all 0.5.x solidity prior to `Solidity 0.5.17`. Specifically, a private function can be overridden in a derived contract by a private function of the same name and types. Fortunately, there is no overriding issue in this code, but we still recommend using `Solidity 0.5.17` or above.

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, as mentioned in Section 2, 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 have analyzed the Ethhmy-bridge design and implementation. The system presents a unique offering in bridging Ethereum and Harmony, two major public blockchains, 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.

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 [6, 7, 8, 9, 11].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [12] 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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

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

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

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

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