SMART CONTRACT AUDIT REPORT

for

FURUCOMBO

Prepared By: Shuxiao Wang

PeckShield

May 4, 2021

Document Properties

Client	Furucombo
Title	Smart Contract Audit Report
Target	Furucombo
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 4, 2021	Xuxian Jiang	Final Release
1.0-rc	May 1, 2021	Xuxian Jiang	Release Candidate
0.3	April 28, 2021	Xuxian Jiang	Additional Findings #2
0.2	April 25, 2021	Xuxian Jiang	Additional Findings #1
0.1	April 16, 2021	Xuxian Jiang	Initial Draft

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 smart contract source code of the **Furucombo** protocol, we outline in this report 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About Furucombo

`Furucombo` is a tool developed for end-users to optimize their DeFi strategy with simple, convenient, and visualized drag-and-drop operations. Specifically, it visualizes complex DeFi protocols into so-called `cubes`. Users setup inputs and outputs as well as the order of the `cubes`, then `Furucombo` bundles all the `cubes` into one transaction and sends out. The protocol names this building-blocks setup a `combo`. The goal of Furucombo is to empower protocol users to optimize and enhance their DeFi strategy with ease, and without knowledge of coding. Given the nature of `Furucombo`, there are numerous combinations and strategies users can make and accomplish.

The basic information of Furucombo is as follows:

Table 1.1: Basic Information of Furucombo

Item	Description
Issuer	Furucombo
Website	https://furucombo.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 4, 2021

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

- https://github.com/dinngodev/furucombo-contract.git (c90dec1)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

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

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

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) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, 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 Logic	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 Furucombo 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	0	
Medium	0	
Low	4	■■■■
Informational	2	■■
Total	6	

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 4 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Furucombo Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Token Returns In Various Handlers	Business Logic	Resolved
PVE-002	Low	Improved flashLoan() Logic in AaveV2 Handler	Business Logic	Resolved
PVE-003	Informational	Credit Delegation Handling Of AaveV2 Handlers	Business Logic	Resolved
PVE-004	Low	Consistency In Token Allowance Approvals	Coding Practices	Resolved
PVE-005	Informational	Payable Uses In Various Handlers	Coding Practices	Resolved
PVE-006	Low	The receive() Support In HandlerBase	Business Logic	Resolved

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 Proper Token Returns In Various Handlers

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: HAaveProtocolV2
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The Furucombo protocol has developed a number of handlers to support combo operations with current popular DeFi protocols, e.g., Aave, Compound, and Uniswap. A number of core APIs, e.g., deposit()/withdraw()/borrow()/repay(), have been abstracted and standardized to facilitate combinations with others. In the following, we examine the AaveV2 support.

Specifically, the function under our analysis is the repay() method. As the name indicates, this method is designed to pay back certain debt for the intended onBehalfOf user. To elaborate, we show below its logic as well as its internal handler _repay().

```
    function repay(
        address asset,
        uint256 amount,
        uint256 rateMode,
        address onBehalfOf
    ) external payable returns (uint256 remainDebt) {
        remainDebt = _repay(asset, amount, rateMode, onBehalfOf);
    }
```

Listing 3.1: HAaveProtocolV2::repay()

```
    function _repay(
        address asset,
        uint256 amount,
        uint256 rateMode,
        address onBehalfOf
```

```
    ) internal returns (uint256 remainDebt) {
        address pool =
            ILendingPoolAddressesProviderV2(PROVIDER).getLendingPool();
        IERC20(asset).safeApprove(pool, amount);

        try
            ILendingPoolV2(pool).repay(asset, amount, rateMode, onBehalfOf)
        {} catch Error(string memory reason) {
            _revertMsg("repay", reason);
        } catch {
            _revertMsg("repay");
        }

        IERC20(asset).safeApprove(pool, 0);

        DataTypes.ReserveData memory reserve =
            ILendingPoolV2(pool).getReserveData(asset);
        remainDebt = DataTypes.InterestRateMode(rateMode) ==
            DataTypes.InterestRateMode.STABLE
            ? IERC20(reserve.stableDebtTokenAddress).balanceOf(onBehalfOf)
            : IERC20(reserve.variableDebtTokenAddress).balanceOf(onBehalfOf);
    }
```

Listing 3.2: HAaveProtocolV2::_repay()

We point out that the given payment `amount` may not be used up and there is a need to return the unused portion back to the user. With that, we suggest to add `_updateToken(asset)` at the end of the `repay()` routine.

The same issue is also applicable to other handlers and their routines, e.g., `HCToken::repayBorrowBehalf()`, `HSCompound::repayBorrow()`, `HBalancerExchange::batchSwapExactOut()`, `HBalancerExchange::multihopBatchSwapExactOut()`, `HBalancerExchange::smartSwapExactOut()`, `HBalancer::joinPool()`, `HUniswap::addLiquidity()`, `HUniswap::tokenToEthSwapOutput()`, `HUniswap::tokenToTokenSwapOutput()`, `HSushiSwap::addLiquidity()`, `HSushiSwap::tokenToEthSwapOutput()`, and `HSushiSwap::tokenToTokenSwapOutput()`.

Recommendation Revise the above-mentioned logic to properly return the unfulfilled payment amount back to the user.

Status This issue has been confirmed. In the meantime, the team clarifies that only when the function call of a handler generates new token, there is a need to update the token to the stack. For the functions that do not use up the token, their token addresses should be updated by another handler that puts the token into the proxy. With that, the team considers no need to address it and plans to leave it as is.

3.2 Improved flashLoan() Logic in AaveV2 Handler

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: HAaveProtocolV2
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

Among various DeFi functionalities and features, flashloan is a disruptive one that allows users to borrow from the reserves within a single transaction, as long as the user returns the borrowed amount plus additional premium. In this section, we examine the flash loan support in Furucombo.

The current protocol supports AaveV2-based flashloans. To elaborate, we show below the flashLoan() routine. This routine implements a rather straightforward logic in validating the given arguments and then invoking the flashLoan() call on the AaveV2 lending pool. At the end, the flashLoan() routine calls _updateToken() to push borrowed assets in the stack so that they will be transferred to the user in the post-process phase.

```
    function flashLoan(
        address[] calldata assets,
        uint256[] calldata amounts,
        uint256[] calldata modes,
        bytes calldata params
    ) external payable {
        if (assets.length != amounts.length) {
            _revertMsg("flashLoan", "assets and amounts do not match");
        }

        if (assets.length != modes.length) {
            _revertMsg("flashLoan", "assets and modes do not match");
        }

        address onBehalfOf = _getSender();
        address pool =
            ILendingPoolAddressesProviderV2(PROVIDER).getLendingPool();

        try
            ILendingPoolV2(pool).flashLoan(
                address(this),
                assets,
                amounts,
                modes,
                onBehalfOf,
                params,
                REFERRAL_CODE
            )
```

```
127         {} catch Error(string memory reason) {
128             _revertMsg("flashLoan", reason);
129         } catch {
130             _revertMsg("flashLoan");
131         }

133         // approve lending pool zero
134         for (uint256 i = 0; i < assets.length; i++) {
135             IERC20(assets[i]).safeApprove(pool, 0);
136             if (modes[i] != 0) _updateToken(assets[i]);
137         }
138     }
```

Listing 3.3: AaveV2::flashLoan()

We notice the call of `_updateToken()` (line 136) is conditioned on `modes[i] != 0`, i.e., only when the funds are being borrowed and there is no need to instantly return the funds before the end of transaction. However, we argue that even when `modes[i] == 0`, there is still a need to call `_updateToken()`. The reason is simply that there is new fund generated, or more precisely borrowed, in this `flashLoan()` handler.

Recommendation Remove the `if`-condition (line 136) to always call `_updateToken()` in `HAaveProtocolV2::flashLoan()`.

Status This issue has been resolved as the team clarifies that in the case of `modes[i] == 0`, the related `assets[i]` will be pushed to stack by other handlers, hence avoiding the need of pushing the same asset in this flashloan handler.

3.3 Credit Delegation Handling Of AaveV2 Handlers

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `HAaveProtocolV2`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The lending platform `Aave2` implements a variety of innovative features. One of them is the so-called `credit delegation`, which in essence allows an user to take uncollateralized loans as long as the user receives delegation from other users that provide the collateral. The feature is mainly implemented with a pair of related routines, i.e., `delegateBorrowAllowance()` and `borrow()`. The `Furucombo` protocol has the built-in support of `Aave2`'s `credit delegation`.

```
    function borrow(
        address asset,
        uint256 amount,
        uint256 rateMode
    ) external payable {
        address onBehalfOf = _getSender();
        _borrow(asset, amount, rateMode, onBehalfOf);
        _updateToken(asset);
    }
```

Listing 3.4: HAaveProtocolV2::borrow()

```
    function _borrow(
        address asset,
        uint256 amount,
        uint256 rateMode,
        address onBehalfOf
    ) internal {
        address pool =
            ILendingPoolAddressesProviderV2(PROVIDER).getLendingPool();

        try
            ILendingPoolV2(pool).borrow(
                asset,
                amount,
                rateMode,
                REFERRAL_CODE,
                onBehalfOf
            )
        {} catch Error(string memory reason) {
            _revertMsg("borrow", reason);
        } catch {
            _revertMsg("borrow");
        }
    }
```

Listing 3.5: HAaveProtocolV2::_borrow()

To elaborate, we show above the related `borrow()` method. It comes to our attention this `credit delegation` feature may require proper attention for its use in `Furucombo`. In particular, to properly use this feature, a `Furucombo` user needs to delegate the borrow allowance to the `Furucombo proxy`. It seems that a malicious actor may take advantage of the delegated borrow allowance to borrow from `AaveV2` and walk way with the borrowed funds at the cost of the legitimate user who delegates the borrow allowance. Fortunately, this issue is eliminated by ensuring the delegate borrow can only occur between the `Furucombo proxy` and the current `msg.sender`.

Recommendation We emphasize that there is no issue in current implementation. And we have intentionally added this issue to ensure that the credit delegation is always properly enforced and isolated.

Status There is no need to address this issue.

3.4 Consistency In Token Allowance Approvals

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [1]

Description

The interaction with various DeFi protocols demands a convenient support to specify or manage the token spending allowance. The Furucombo protocol currently supports two patterns of token spending approval. The first pattern is a traditional one in making use of IERC20's safeApprove() to approve the spending right before the interaction and then revoking the allowance immediately after. The second pattern is the use of a customized helper _tokenApprove(). To elaborate, we show below this helper routine.

```
    function _tokenApprove(
        address token,
        address spender,
        uint256 amount
    ) internal {
        try IERC20Usdt(token).approve(spender, amount) {} catch {
            IERC20(token).safeApprove(spender, 0);
            IERC20(token).safeApprove(spender, amount);
        }
    }
```

Listing 3.6: HandlerBase::_tokenApprove()

The first pattern is more fine-grained in controlling the exact allowance amount while the second pattern ensures this time's spending, without explicitly resetting the allowance to 0. Note that the second pattern may come with slight lower gas cost when compared with the first one. From the maintenance perspective, it is suggested to be consistent in specifying the token allowance, instead of using two patterns in a mixed way.

Recommendation Be consistent in specifying the token allowance among current handlers.

Status This issue has been resolved. The presence of the second pattern with _tokenApprove() is to accommodate certain ERC20 tokens, e.g., HBTC, that may not allow set allowance to zero.

3.5 Payable Uses In Various Handlers

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [1]

Description

As mentioned in Section 3.1, the Furucombo protocol has developed a number of handlers to support combo operations with current popular DeFi protocols, and a number of core APIs, e.g., deposit() and withdraw(), have been abstracted and standardized to facilitate the combinations with others. In the following, we examine the WETH support.

The WETH integration is supported by the HWeth contract. To elaborate, we show below its implementation. The contract has two main routines deposit() and withdraw(). The first routine allows to deposit Ether into the handler for WETH-wrapping while the second routine withdraws Ether from the handler. It comes to our attention that the withdraw() routine is defined to be payable, meaning this routine may take Ether as input. However, this is apparently not the case. With that, we suggest to remove this payable keyword from the withdraw() routine.

```
contract HWeth is HandlerBase {
    // prettier-ignore
    address payable public constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    function getContractName() public pure override returns (string memory) {
        return "HWeth";
    }

    function deposit(uint256 value) external payable {
        try IWETH9(WETH).deposit{value: value}() {} catch Error(
            string memory reason
        ) {
            _revertMsg("deposit", reason);
        } catch {
            _revertMsg("deposit");
        }
        _updateToken(WETH);
    }

    function withdraw(uint256 wad) external payable {
        try IWETH9(WETH).withdraw(wad) {} catch Error(string memory reason) {
            _revertMsg("withdraw", reason);
        } catch {
            _revertMsg("withdraw");
        }
```

```
    }
}
```

Listing 3.7: The HWeth Handler

The same issue is applicable to a number of other handlers and routines. Examples include the following routines from HAaveProtocolV2: deposit(), withdraw(), withdrawETH(), repay(), borrow(), and borrowETH().

Recommendation Avoid using the payable keyword when there is no need.

Status This issue has been resolved. And the reason why using payable even for non-payable routines is due to the fact that current handlers are invoked via delegatecall, which keeps msg.sender and msg.value unchanged. Since there may be multiple actions within a single batchExec(), if any action needs Ether with non-zero msg.value, the same msg.value will be applied to all delegatecall'ed handlers even for those don't use Ether.

3.6 The receive() Support In HandlerBase

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: HandlerBase
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In last Section 3.5, we have examined the HWeth handler that provides the support of Ether wrapping and unwrapping. Note that all current handlers are inherited from a common base, i.e., HandlerBase. It is interesting to notice that though handlers have defined a number of combo routines that can take the Ether input, the current base HandlerBase does not support the receive() method.

To elaborate, we show below again the HWeth handler. Note the withdraw() unwraps WETHs back into Ether. As the HandlerBase contract has not defined the fallback routine, we suggest the need of adding a receive() method. Note this receive keyword is introduced since Solidity $0.6.x$ in order to make contracts more explicit when their fallback functions are called. In particular, the receive() method is used as a fallback function in a contract and is called when Ether is sent to a contract with no calldata. If the receive() method does not exist, it will use the default fallback function.

```
contract HWeth is HandlerBase {
    // prettier-ignore
    address payable public constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

```

```
    function getContractName() public pure override returns (string memory) {
        return "HWeth";
    }

    function deposit(uint256 value) external payable {
        try IWETH9(WETH).deposit{value: value}() {} catch Error(
            string memory reason
        ) {
            _revertMsg("deposit", reason);
        } catch {
            _revertMsg("deposit");
        }
        _updateToken(WETH);
    }

    function withdraw(uint256 wad) external payable {
        try IWETH9(WETH).withdraw(wad) {} catch Error(string memory reason) {
            _revertMsg("withdraw", reason);
        } catch {
            _revertMsg("withdraw");
        }
    }
}
```

Listing 3.8: The `HWeth` Handler

Recommendation Add the `receive()` function as there is no default fallback routine in `HandlerBase`.

Status While the above suggestion may sound reasonable, it turns out that these handlers are not invoked directly. As mentioned earlier, in `Furucombo`, they are invoked via `delegatecall` by the `Furucombo proxy`. And the proxy has already defined the `receive()` function. With that, this issue becomes irrelevant and there is no need to be addressed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Furucombo` protocol. The system presents a unique offering as a trustless mechanism to empower protocol users to optimize and enhance their DeFi strategy with ease, and without knowledge of coding. The current code base is well organized and those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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