



# SMART CONTRACT AUDIT REPORT

for

## BONDING FINANCE



Prepared By: Xiaomi Huang

PeckShield  
April 30, 2023

## Document Properties

Client	Bonding Finance
Title	Smart Contract Audit Report
Target	Bonding Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	April 30, 2023	Xuxian Jiang	Final Release
1.0-rc1	April 29, 2023	Xuxian Jiang	Release Candidate #1

## Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Bonding Finance . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Incorrect Reward Calculation in MasterChef . . . . .	11
3.2	Staking Incompatibility With Deflationary Tokens . . . . .	12
3.3	Duplicate Pool Detection and Prevention in MasterChef . . . . .	14
3.4	Suggested Adherence Of Checks-Effects-Interactions Pattern . . . . .	15
3.5	Trust Issue of Admin Keys . . . . .	16
<b>4</b>	<b>Conclusion</b>	<b>18</b>
	<b>References</b>	<b>19</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Bonding Finance` protocol, we outline in the 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 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 Bonding Finance

With the rise of LSD's like `stETH`, `frxETH`, `rETH`, and many others, one piece that's missing from the picture is tokenizing the ETH yield APR. Holders of LSDs may want to sell their future yield for an upfront payment, shorting the future interest rate. On the other hand, some may want to bet on the ETH yield rising in the future. `Bonding Finance` aims to allow users to separate their LSD token into two parts, an intrinsic part that represents 1 ETH (`dToken`) and an extrinsic part that represents the future yield in perpetuity (`yToken`). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Bonding Finance

Item	Description
Name	Bonding Finance
Website	<a href="https://bonding.finance/">https://bonding.finance/</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 30, 2023

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

- <https://github.com/bonding-finance/bonding-finance-contracts.git> (0505fa9)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/bonding-finance/bonding-finance-contracts.git> (dc04251)

## 1.2 About PeckShield

PeckShield Inc. [9] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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 checklist of 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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 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
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logic</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 implementation of the `Bonding Finance` protocol. 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 logic, 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	2	
Low	3	
Informational	0	
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 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Bonding Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Reward Calculation in MasterChef	Business Logic	Resolved
PVE-002	Low	Staking Incompatibility With Deflationary Tokens in MasterChef	Business Logic	Resolved
PVE-003	Low	Duplicate Pool Detection and Prevention in MasterChef	Business Logic	Resolved
PVE-004	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-005	Medium	Trust on Admin Keys	Security Features	Mitigated

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 Incorrect Reward Calculation in MasterChef

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

#### Description

The Bonding Finance protocol provides an incentive mechanism that rewards the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool. While examining the logic to update staking pools, we notice an issue during the reward calculation.

To elaborate, we show below the related `updatePool()` function, which is designed to refresh the reward for the given pool. However, it comes to our attention that the reward is currently computed as `reward = (rewardPerBlock * pool.allocPoint) / totalAllocPoint` (line 110), which does not take into the elapsed blocks from `pool.lastRewardBlock` to the latest `block.number`. Note that the same issue is also applicable to the `pendingRewards()` routine.

```

100     function updatePool(uint256 _pid) public override {
101         PoolInfo storage pool = poolInfo[_pid];
102         if (block.number <= pool.lastRewardBlock) {
103             return;
104         }
105         uint256 stakedSupply = ERC20(pool.token).balanceOf(address(this));
106         if (stakedSupply == 0) {
107             pool.lastRewardBlock = block.number;
108             return;
109         }
110         uint256 reward = (rewardPerBlock * pool.allocPoint) / totalAllocPoint;
111         pool.accRewardsPerShare += (reward * 1e18) / stakedSupply;

```

```

112     pool.lastRewardBlock = block.number;
113 }

```

Listing 3.1: MasterChef::updatePool()

**Recommendation** Improve the above-mentioned routines to properly compute the right reward for distribution.

**Status** The issue has been fixed in this commit: [bd326ed](#).

## 3.2 Staking Incompatibility With Deflationary Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

In the Bonding Finance protocol, the MasterChef contract is designed to allow users to stake and earn rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e., `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract uses the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```

55     function deposit(uint256 _pid, uint256 _amount) external override {
56         PoolInfo storage pool = poolInfo[_pid];
57         UserInfo storage user = userInfo[_pid][msg.sender];
58         updatePool(_pid);
59         if (user.amount > 0) {
60             uint256 pending = ((user.amount * pool.accRewardsPerShare) / 1e18) - user.
                rewardDebt;
61             _transferRewards(msg.sender, pending);
62         }
63         ERC20(pool.token).safeTransferFrom(msg.sender, address(this), _amount);
64         user.amount += _amount;
65         user.rewardDebt = (user.amount * pool.accRewardsPerShare) / 1e18;
66
67         emit Deposit(msg.sender, _pid, _amount);
68     }

```

Listing 3.2: MasterChef::deposit()

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accRewardsPerShare` via dividing `reward` by `stakedSupply`, where the `stakedSupply` is derived from `ERC20(pool.token).balanceOf(address(this))` (line 105). Because the balance inconsistencies of the pool, the `stakedSupply` could be 1 Wei and thus may yield a huge `pool.accRewardsPerShare` as the final result, which dramatically inflates the pool's reward.

```

100     function updatePool(uint256 _pid) public override {
101         PoolInfo storage pool = poolInfo[_pid];
102         if (block.number <= pool.lastRewardBlock) {
103             return;
104         }
105         uint256 stakedSupply = ERC20(pool.token).balanceOf(address(this));
106         if (stakedSupply == 0) {
107             pool.lastRewardBlock = block.number;
108             return;
109         }
110         uint256 reward = (rewardPerBlock * pool.allocPoint) / totalAllocPoint;
111         pool.accRewardsPerShare += (reward * 1e18) / stakedSupply;
112         pool.lastRewardBlock = block.number;
113     }

```

Listing 3.3: `MasterChef::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

**Recommendation** Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like

features kicked in later, which should be verified carefully.

**Status** This issue has been resolved as the team confirms no deflationary tokens will be supported.

### 3.3 Duplicate Pool Detection and Prevention in MasterChef

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

#### Description

The Bonding Finance protocol provides an incentive mechanism that rewards the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its  $\text{allocPoint} \times 100\% / \text{totalAllocPoint}$  share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded governance tokens and more can be scheduled for addition. To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```

119     function add(
120         uint256 _allocPoint,
121         address _token,
122         bool _withUpdate
123     ) external override onlyOwner {
124         if (_withUpdate) {
125             massUpdatePools();
126         }
127
128         uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
129         totalAllocPoint += _allocPoint;
130         poolInfo.push(
131             PoolInfo({
132                 token: _token,

```

```
133         allocPoint: _allocPoint,
134         lastRewardBlock: lastRewardBlock,
135         accRewardsPerShare: 0
136     })
137 };
138 }
```

Listing 3.4: `MasterChef::add()`

**Recommendation** Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate. We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers. Moreover, it is also applicable to validate the paired strategy is not duplicated!

**Status** The issue has been fixed in this commit: 3615891.

## 3.4 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

### Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 87) starts before effecting the update on the internal state (lines 89-90), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

84     function emergencyWithdraw(uint256 _pid) external override {
85         PoolInfo storage pool = poolInfo[_pid];
86         UserInfo storage user = userInfo[_pid][msg.sender];
87         ERC20(pool.token).safeTransfer(msg.sender, user.amount);
88         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
89         user.amount = 0;
90         user.rewardDebt = 0;
91     }

```

Listing 3.5: MasterChef::emergencyWithdraw()

Note that other routines share the same issue, including `deposit()` and `withdraw()` from the same contract as well as `PerpetualBondStaking::collectSurplus()` and `PerpetualBondVault::collectFee()`.

**Recommendation** Strictly follow the best practice of the checks-effects-interactions pattern or make use of the `nonReentrant` modifier to block possible re-entrancy.

**Status** The issue has been fixed in this commit: [e80e389](#).

## 3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

### Description

In the Bonding Finance protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, create/manage new vaults, as well as set up rewards). In the following, we show the representative functions potentially affected by the privilege of the account.

```

43     function setStaking(address vault, address staking) external override onlyOwner {
44         require(staking != address(0));
45         require(IPerpetualBondStaking(staking).vault() == vault, "!valid");
46
47         IPerpetualBondVault(vault).setStaking(staking);
48     }
49

```



```

50     function collectFees(address vault) external override onlyOwner {
51         require(vault != address(0));
52         require(feeInfo.feeTo != address(0), "feeTo is 0");
53
54         IPerpetualBondVault(vault).collectFees(feeInfo.feeTo);
55     }
56
57     function collectSurplus(address staking) external override onlyOwner {
58         require(staking != address(0));
59         require(feeInfo.feeTo != address(0), "feeTo is 0");
60
61         IPerpetualBondStaking(staking).collectSurplus(feeInfo.feeTo);
62     }
63
64     function setFeeTo(address feeTo) external override onlyOwner {
65         feeInfo.feeTo = feeTo;
66     }
67
68     function setFee(uint256 fee) external override onlyOwner {
69         require(fee <= 100, "Fee > 100");
70
71         feeInfo.fee = fee;
72     }

```

Listing 3.6: Example Privileged Operations in PerpetualBondFactory

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

**Recommendation** Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

**Status** The issue has been confirmed by the team. The team intends to introduce multi-sig and timelock mechanisms to mitigate this issue.

## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Bonding Finance` protocol, which aims to allow users to separate their `LSD` token into two parts, an intrinsic part that represents 1 ETH (`dToken`) and an extrinsic part that represents the future yield in perpetuity (`yToken`). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [6] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.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](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.

- [11] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

