



SMART CONTRACT AUDIT REPORT

for

COVERFORGE



Prepared By: Shuxiao Wang

PeckShield
February 26, 2021

Document Properties

Client	CoverForge
Title	Smart Contract Audit Report
Target	CoverForge
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 26, 2021	Xuxian Jiang	Final Release
1.0-rc	February 25, 2021	Xuxian Jiang	Release Candidate #1
0.2	February 23, 2021	Xuxian Jiang	Additional Findings
0.1	February 18, 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	4
1.1	About CoverForge	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	11
3.2	Suggested Adherence of Checks-Effects-Interactions	13
3.3	Possible Costly CoverForge From Improper Initialization	15
3.4	Potential Front-Running/MEV With Reduced Buyback	16
4	Conclusion	18
	References	19

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the CoverForge 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 CoverForge

Cover allows DeFi users to be protected against smart contract risk. It stabilizes the dynamic and even turbulent DeFi space by instilling confidence and trust between protocols and their users. By bridging the gap between decentralized finance and traditional finance, Cover aims to open the doors of DeFi to all investors. Technically, it is designed to allow all smart contracts to be covered up to the Total Value Locked (TVL) in the covered smart contract, and also allows the market to set coverage prices as opposed to a bonding curve. The audited CoverForge allows for receiving Cover fee to buy back COVER tokens and bringing additional incentivizes to the Cover community.

The basic information of the CoverForge protocol is as follows:

Table 1.1: Basic Information of The CoverForge Protocol

Item	Description
Issuer	CoverForge
Website	https://www.coverprotocol.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 26, 2021

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

- <https://github.com/CoverProtocol/cover-forge.git> (4315d89)

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

- <https://github.com/CoverProtocol/cover-forge.git> (ceecec5)

1.2 About PeckShield

PeckShield Inc. [11] 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	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 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 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 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 1.3: 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




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 CoverForge 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	1	
Low	2	
Informational	1	
Total	4	

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 that 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 1 medium-severity vulnerability, and 2 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key CoverForge Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Safe-Version Replacement With <code>safeApprove()</code> , <code>safeTransfer()</code> And <code>safeTransferFrom()</code>	Coding Practices	Fixed
PVE-002	Informational	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-003	Low	Possible Costly CoverForge From Improper Initialization	Time and State	Confirmed
PVE-004	Low	Potential Front-running/MEV With Reduced Buyback	Time and State	Fixed

Besides recommending specific countermeasures to mitigate these issues, based on 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 0.8.0` instead of specifying a range, e.g., `pragma solidity ^0.8.0`.

In addition, we also emphasize that 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. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Safe-Version Replacement With `safeApprove()`, `safeTransfer()` And `safeTransferFrom()`

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: CoverFeeReceiver
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (<https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>).

```

194  /**
195  * @dev Approve the passed address to spend the specified amount of tokens on behalf
      of msg.sender.
196  * @param _spender The address which will spend the funds.
197  * @param _value The amount of tokens to be spent.
198  */
199  function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201      // To change the approve amount you first have to reduce the addresses'
202      // allowance to zero by calling 'approve(_spender, 0)' if it is not
203      // already 0 to mitigate the race condition described here:

```

```

204 // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205 require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

207 allowed[msg.sender][_spender] = _value;
208 Approval(msg.sender, _spender, _value);
209 }

```

Listing 3.1: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

More importantly, the `approve()` function does not have a return value. However, the IERC20 interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount) external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of USDT's `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using `SafeERC20` for IERC20. Similarly, there is a safe version of `transfer()/transferFrom()` as well, i.e., `safeTransfer()/safeTransferFrom()`.

In the following, we show the `buyBack()` routine in the `CoverFeeReceiver` contract. If the USDT token is supported as `_token`, the unsafe version of `_token.approve(address(_router), type(uint256).max)` (line 41) may revert as there is no return value in the USDT token contract's `approve()` implementation (but the IERC20 interface expects a return value)!

```

27 function buyBack(IERC20 _token, IRouter _router, address[] calldata _path, uint256
   _maxSwapAmt) external override onlyOwner {
28     require(_path[0] == address(_token), "input token != _token");
29     require(_path[0] != cover, "input token cannot be COVER");
30     require(_path[_path.length - 1] == cover, "output token != COVER");
31     require(_maxSwapAmt > 0, "_maxSwapAmt <= 0");
32     uint256 balance = _token.balanceOf(address(this));
33     require(balance > 0, "_token balance is 0");
34     uint256 swapAmt = balance < _maxSwapAmt ? balance : _maxSwapAmt;
35     if (feeNumToTreasury > 0) {
36         uint256 amtToTreasury = swapAmt * feeNumToTreasury / 10000;
37         _token.transfer(treasury, amtToTreasury);
38         swapAmt = swapAmt - amtToTreasury;
39     }
40     if (_token.allowance(address(this), address(_router)) < swapAmt) {
41         _token.approve(address(_router), type(uint256).max);
42     }

```

```

43     _router.swapExactTokensForTokens(swapAmt, 0, _path, forge, block.timestamp + 1
        hours);
44     emit BuyBack(_token, swapAmt);
45 }

```

Listing 3.2: CoverFeeReceiver::buyBack()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been fixed by this commit: [ceecec5](#).

3.2 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: CoverForge
- Category: Time and State [7]
- CWE subcategory: CWE-663 [3]

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 [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the CoverForge as an example, the `deposit()` 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 28) starts before effecting the update on internal states (lines 30 – 35), 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 very same `deposit()` function.

```

26     function deposit(uint256 _amount) public override {
27         uint256 totalCover = cover.balanceOf(address(this));

```

```

28     cover.transferFrom(msg.sender, address(this), _amount);
29     uint256 totalShares = totalSupply();
30     if (totalShares == 0 totalCover == 0) {
31         _mint(msg.sender, _amount);
32     } else {
33         uint256 myShare = _amount * totalShares / totalCover;
34         _mint(msg.sender, myShare);
35     }
36     emit Deposit(msg.sender, _amount);
37 }

```

Listing 3.3: CoverForge::deposit()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions best practice. An example revision on the emergencyWithdraw routine is shown below:

```

26     function deposit(uint256 _amount) public override {
27         uint256 totalCover = cover.balanceOf(address(this));
28         uint256 totalShares = totalSupply();
29         if (totalShares == 0 totalCover == 0) {
30             _mint(msg.sender, _amount);
31         } else {
32             uint256 myShare = _amount * totalShares / totalCover;
33             _mint(msg.sender, myShare);
34         }
35         cover.transferFrom(msg.sender, address(this), _amount);
36         emit Deposit(msg.sender, _amount);
37     }

```

Listing 3.4: Revised CoverForge::deposit()

Status The issue has been fixed by this commit: 9d96805.

3.3 Possible Costly CoverForge From Improper Initialization

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: CoverFeeReceiver
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

The CoverForge protocol allows users to deposit COVER tokens and get in return xCOVER to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token, i.e., xCOVER, extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This routine is used for participating users to deposit the supported COVER and get respective xCOVER pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```

26     function deposit(uint256 _amount) public override {
27         uint256 totalCover = cover.balanceOf(address(this));
28         cover.transferFrom(msg.sender, address(this), _amount);
29         uint256 totalShares = totalSupply();
30         if (totalShares == 0 || totalCover == 0) {
31             _mint(msg.sender, _amount);
32         } else {
33             uint256 myShare = _amount * totalShares / totalCover;
34             _mint(msg.sender, myShare);
35         }
36         emit Deposit(msg.sender, _amount);
37     }

```

Listing 3.5: CoverForge::deposit()

Specifically, when the pool is being initialized (line 30), the share value directly takes the value of `amount` (line 31), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `myShare = _amount = 1 WEI`. With that, the actor can further deposit a huge amount of COVER assets with the goal of making the xCOVER pool token extremely expensive.

An extremely expensive xCOVER pool token can be very inconvenient to use as a small number of 1WEI may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `uniswap`. When providing the initial liquidity to the contract (i.e. when `totalSupply` is 0), the liquidity provider must sacrifice 1000 LP

tokens (by sending them to `address(0)`). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the share amount when the pool is being initialized. Or safeguard the first deposit to avoid being manipulated.

Status This issue has been confirmed and the team will exercise extra caution in having a guarded launch to ensure the pool will be properly initialized.

3.4 Potential Front-Running/MEV With Reduced Buyback

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `CoverFeeReceiver`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [4]

Description

As common in various DeFi protocols, there is a constant need to convert from one token to another. The `CoverForge` protocol has a `buyBack` routine that is designed to buy back `COVER` with available tokens. To elaborate, we show below the `buyBack()` routine.

```

27     function buyBack(IERC20 _token, IRouter _router, address[] calldata _path, uint256
    _maxSwapAmt) external override onlyOwner {
28         require(_path[0] == address(_token), "input token != _token");
29         require(_path[0] != cover, "input token cannot be COVER");
30         require(_path[_path.length - 1] == cover, "output token != COVER");
31         require(_maxSwapAmt > 0, "_maxSwapAmt <= 0");
32         uint256 balance = _token.balanceOf(address(this));
33         require(balance > 0, "_token balance is 0");
34         uint256 swapAmt = balance < _maxSwapAmt ? balance : _maxSwapAmt;
35         if (feeNumToTreasury > 0) {
36             uint256 amtToTreasury = swapAmt * feeNumToTreasury / 10000;
37             _token.transfer(treasury, amtToTreasury);
38             swapAmt = swapAmt - amtToTreasury;
39         }
40         if (_token.allowance(address(this), address(_router)) < swapAmt) {
41             _token.approve(address(_router), type(uint256).max);
42         }
43         _router.swapExactTokensForTokens(swapAmt, 0, _path, forge, block.timestamp + 1
            hours);
44         emit BuyBack(_token, swapAmt);
45     }

```

Listing 3.6: `CoverFeeReceiver::buyBack()`

We notice the conversion is routed to `UniswapV2` in order to swap one token to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of buyback amount.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of farming users.

Status The issue has been fixed by this commit: `ceecec5`.



4 | Conclusion

In this audit, we have analyzed the design and implementation of `CoverForge`, which is built to seamlessly work with `COVER` protocol, which serves the purpose of providing much-needed insurance coverage to blockchain smart contracts against possible exploits and hacks. During the audit, we notice that 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.



References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). <https://cwe.mitre.org/data/definitions/362.html>.
- [3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [4] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [5] MITRE. CWE CATEGORY: 7PK - Time and State. <https://cwe.mitre.org/data/definitions/361.html>.
- [6] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [7] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.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. PeckShield Inc. <https://www.peckshield.com>.
- [12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [13] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

