SMART CONTRACT AUDIT REPORT

for

Shield Protocol

Prepared By: Yiqun Chen

PeckShield

November 26, 2021

Document Properties

Client	Shield
Title	Smart Contract Audit Report
Target	Shield
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 26, 2021	Xuxian Jiang	Final Release
1.0-rc1	November 19, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

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

`Shield` is a non-custodial decentralized derivative exchange that trades risk-free perpetual contracts. The risk-free perpetual contract is the solution from `Shield` to the existing limitations within the decentralized derivative ecosystem. It uses a combination of 0 position loss, a dual liquidity pool model, high leverages, a decentralized brokerage system, and external liquidators to counteract the existing limitations. This new perpetual product goes above and beyond the mentioned limitations in the current derivative products, aiming to get to be a more competitive space and bring DeFi the next generation of global decentralized derivative infrastructure.

The basic information of Shield is as follows:

Table 1.1: Basic Information of Shield

Item	Description
Name	Shield
Website	https://shieldex.io/
Type	Ethereum and BSC Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 26, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the audited protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/ShieldDAODev/shield-contracts-audit.git (32e0097)

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

- https://github.com/ShieldDAODev/shield-contracts-audit.git (a9abdcb)

1.2 About PeckShield

PeckShield Inc. [14] 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 the OWASP Risk Rating Methodology [13]:

- 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 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) [12], 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

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Shield` 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	1	■
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 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Shield Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Logic In SLDBroker::calcBrokerAmount()	Business Logic	Fixed
PVE-002	Informational	Proper Decimals In SLDBuyBack::getTotalFiatTokenAmount()	Numeric Errors	Confirmed
PVE-003	Low	Improved Precision By Multiplication And Division Reordering	Coding Practices	Fixed
PVE-004	Low	Improved Corner Case Handling in Fraction::sqrt()	Coding Practices	Fixed
PVE-005	Low	Consistent Reentrancy Protection in SLDOption	Time and State	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Logic In SLDBroker::calcBrokerAmount()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SLDBroker
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

To lower the barrier for protocol users, the Shield protocol has the support of so-called brokers. Implemented in the SLDBroker contract, the brokers can be ranked and incentivized to better engage protocol users. While reviewing the broker support, we notice an internal function needs to be improved.

To elaborate, we show below this function, i.e., calcBrokerAmount(). As the name indicates, this function is designed for calculating ranks and rewards for brokers. However, the calculation for the USDT-related rewards mistakenly uses the entire trading volume as the reward without taking into account the proper level-specific RewardNumerator. As a result, current brokers at the ranking level B will received more rewards than expected, at the potential loss of other protocol brokers!

```
    function calcBrokerAmount(
        address _broker,
        uint256 _tokenType,
        uint256 _amount
    ) internal {
        // current ranking for broker
        uint256 originalRating = brokersRating[_broker];
        brokerInvitedAmount[_broker] = brokerInvitedAmount[_broker].add(
            _amount
        );

        uint256 index;
        index = (originalRating > BROKERSLENGTH)
```

```
            ? (BROKERSLENGTH)
            : (originalRating - 1);
        for (uint256 i = index; i > 0; i--) {
            if (
                brokerInvitedAmount[_broker] <=
                brokerInvitedAmount[brokersRatingList[i - 1]]
            ) {
                break;
            }
            // swap adjacent rankings
            address tmpPriviousBroker = brokersRatingList[i - 1];
            (
                brokersRating[tmpPriviousBroker],
                brokersRating[_broker],
                brokersRatingList[i],
                brokersRatingList[i - 1]
            ) = (i + 1, i, tmpPriviousBroker, _broker);
        }

        uint256 currentRating = brokersRating[_broker];
        if (currentRating <= BROKERSRATINGA) {
            // Ranking Level A
            uint256 rewards = _amount.mul(LevelARewardNumerator).div(
                RewardPortionDenominator
            );
            brokersRewards[_broker][_tokenType]
                .brokersClaimRewards = brokersRewards[_broker][_tokenType]
                .brokersClaimRewards
                .add(rewards);
            emit CalcBrokerAmountA(
                _broker,
                currentRating,
                _amount,
                rewards,
                brokersRewards[_broker][_tokenType].brokersClaimRewards
            );
        } else if (currentRating <= BROKERSRATINGB) {
            // Ranking Level B
            uint256 rewards = _amount.mul(LevelBRewardNumerator).div(
                RewardPortionDenominator
            );
            brokersRewards[_broker][_tokenType]
                .brokersClaimRewards = brokersRewards[_broker][_tokenType]
                .brokersClaimRewards
                .add(_amount);
            emit CalcBrokerAmountB(
                _broker,
                currentRating,
                _amount,
                rewards,
                brokersRewards[_broker][_tokenType].brokersClaimRewards
            );
```

```
        } else if (currentRating <= BROKERSLENGTH) {
        ...
    }
```

Listing 3.1: `SLDBroker::calcBrokerAmount()`

Recommendation Revise the above `calcBrokerAmount()` function to properly compute the broker rewards.

Status This issue has been fixed in this commit: `b5eb811`.

3.2 Proper Decimals In SLDBuyBack::getTotalFiatTokenAmount()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SLDBuyBack`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [2]

Description

In `Shield`, there is a `SLDBuyBack` contract that is designed to buy back `SLD` tokens and reduce the circulation amount. The contract supports three stable coins `DAI`, `USDT`, and `USDC` and provides a public function to compute the buyback price.

To illustrate, we show below this buyback function. This function has the need of determining the current total stable coins in the buyback pool (via `getTotalFiatTokenAmount()`). When examining the `getTotalFiatTokenAmount()` logic, we notice the implicit requirement of three stable coins sharing the same decimals, which may not be the case in current blockchains, such as `Ethereum mainnet`. Note that the decimal plays a critical role to compute the buyback price and the associated value.

```
    /**
     * @dev Get buyback price.
     * @return _price buyback price.
     */
    function getBuybackPrice() public view returns (uint256 _price) {
        uint256 totalFiatAmount = getTotalFiatTokenAmount();
        uint256 remainingSLD = buybackInfo[roundID].remaining;

        _price = totalFiatAmount.mul(priceDecimal).div(remainingSLD);
    }

    /**
     * @dev Get total fiat token amount in buyback pool.
```

```
     * @return _amount total fiat token amount.
     */
    function getTotalFiatTokenAmount() public view returns (uint256 _amount) {
        _amount = DAIToken
            .balanceOf(address(this))
            .add(USDTToken.balanceOf(address(this)))
            .add(USDCToken.balanceOf(address(this)));
    }
```

Listing 3.2: SLDBuyBack::getBuybackPrice()/getTotalFiatTokenAmount()

Specifically, the implicit assumption of sharing the same decimals among the supported stable coins may need to strictly enforced, which is currently missing. The use of different decimals may lead to unexpected results when there is a need to compute the buyback price.

Recommendation Enforce the implicit assumption by ensuring the same decimals among supported stable coins.

Status The issue has been confirmed.

3.3 Improved Precision By Multiplication And Division Reordering

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Aggregator`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.1, `SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, if we examine the `Aggregator::latestRoundData()` routine, this routine has internal computation of `latestPrice` that involves a number of operators, i.e., `uint256(10**uint256(decimals)).div(uint256(price)).mul((10**uint256(decimals)))` (lines 56−58). This computation can be revised as `uint256(10**uint256(decimals)).mul((10**uint256(decimals)).div(uint256(price))` to reduce the precision loss.

```
    function latestRoundData() external view returns (uint256, uint8) {
        uint8 decimals = aggregator.decimals();
        (, int256 price, , , ) = aggregator.latestRoundData();
        uint256 latestPrice = uint256(price);
        if (reverse) {
            latestPrice = uint256(10**uint256(decimals))
                .div(uint256(price))
                .mul((10**uint256(decimals)));
        }
        return (latestPrice, decimals);
    }
```

Listing 3.3: Aggregator::latestRoundData()

A similar optimization can also be applicable to the calculation of the `DateTime::getIntervalPeriods()` function. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in this commit: `de3a1eb`.

3.4 Improved Corner Case Handling in Fraction::sqrt()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SLDFraction`
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [4]

Description

The `Shield` protocol has developed an interesting derivative product, which has a constant need of calculating the integer square root of a given number, i.e., the familiar `sqrt()` function. The `sqrt()` function, implemented in `SLDFraction`, follows the `Babylonian` method for calculating the integer square root. Specifically, for a given x, we need to find out the largest integer z such that $z^2 <= x$.

```
    function sqrt(uint256 x) internal pure returns (uint256 y) {
        uint256 z = (x + 1) / 2;
        y = x;
        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
```

```
320     }
```

Listing 3.4: `SLDFraction::sqrt()`

We show above current `sqrt()` implementation. The initial value of z to the iteration was given as $z = (x+1)/2$, which results in an integer overflow when $x = uint256(-1)$. In other words, the overflow essentially sets z to zero, leading to a `division by zero` in the calculation of $z = (x/z + z)/2$ (line 25).

Note that this does not result in an incorrect return value from `sqrt()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a `divide by zero`, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss.

To address this particular corner case, We suggest to change the initial value to $z = x/2 + 1$, making `sqrt()` well defined over its all possible inputs.

Recommendation Revise the above calculation to avoid the unnecessary integer overflow.

Status This issue has been fixed in this commit: `10249b8`.

3.5 Consistent Reentrancy Protection in SLDOption

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SLDOption`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

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` [16] exploit, and the recent `Uniswap/Lendf.Me` hack [15].

We notice there is an occasions where the `checks-effects-interactions` principle is violated. Using the `SLDOption` as an example, the `deposit()` function (see the code snippet below) is provided to deposit additional tokens into the option contract. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

In particular, the interaction with the external contract inside `deposit()` (line 106) starts before effecting the update on the internal state, 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.

```
103     function deposit(uint256 _amount) public {
104         require(_amount >= minDepositAmount, "too small");
105 
106         _safeTransferFrom(tokenAddr, msg.sender, address(this), _amount);
107 
108         AccountInfo storage userAcc = userAccount[msg.sender];
109         userAcc.depositAmount = userAcc.depositAmount.add(_amount);
110         userAcc.availableAmount = userAcc.availableAmount.add(_amount);
111 
112         emit SLDDeposit(msg.sender, address(this), _amount);
113 
114         emit BalanceOfTaker(
115             msg.sender,
116             userAcc.depositAmount,
117             userAcc.availableAmount,
118             userAcc.liquidationFee
119         );
120     }
```

Listing 3.5: `SLDOption::deposit()`

Note that other functions in the same contract has the proper `lock` modifier to prevent potential `re-entrancy`. However, this `lock` modifier is currently missing in the above `deposit()` function. It is therefore suggested to apply the same modifier to it as well.

Recommendation Apply necessary reentrancy prevention by utilizing the `lock` modifier to the above-mentioned function.

Status This issue has been fixed in this commit: `6487da5`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Shield` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., fee adjustment, and parameter setting). It also has the privilege to regulate or govern the flow of assets for borrowing and lending among the involved components, i.e., `SLDOption`, `SLDBuyBack`, and `SLDReward`.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Shield` protocol.

```
    function setPriAndPubPool(address _priPool, address _pubPool)
        public
        onlyOwner
    {
        require(
            address(_priPool) != address(0x0) &&
                address(_pubPool) != address(0x0),
            "ZERO"
        );
        privPool = IPrivatePool(_priPool);
        pubPool = IPublicPool(_pubPool);
    }

    /**
     * @dev Set risk fund address
     * @param _riskFundAddr Risk fund address.
     */
    function setRiskFundAddr(address _riskFundAddr) public onlyOwner {
        require(address(_riskFundAddr) != address(0x0), "ZERO");
        riskFundAddr = _riskFundAddr;
    }

    /**
     * @dev Set broker contract address
     * @param _brokerAddr Broker contract address.
     */
    function setBrokerAddr(address _brokerAddr) public onlyOwner {
        require(address(_brokerAddr) != address(0x0), "ZERO");
        brokerAddr = _brokerAddr;
    }

    /**
     * @dev Set liquidation contract address
     * @param _liquidatorAddr Liquidation contract address.
     */
    function setLiquidatorAddr(address _liquidatorAddr) public onlyOwner {
        require(address(_liquidatorAddr) != address(0x0), "ZERO");
        liquidatorAddr = _liquidatorAddr;
    }
```

Listing 3.6: Various `Setters` in `SLDOption`

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the `owner` will be managed by a timelock contract.

We point out that a compromised `owner` account is capable of modifying current protocol configuaration with adverse consequences, including permanent lock-down of user funds.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract.

Status This issue has been confirmed. The team clarifies that the `owner` privilege will be transferred to an eventual `DAO`-like governance contract.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Shield` protocol, which is a unique, robust offering as a decentralized perpetual contracts for crypto derivatives trading. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

[15] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

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