SMART CONTRACT AUDIT REPORT

for

Lemma Stablecoin

Prepared By: Yiqun Chen

PeckShield

October 8, 2021

Document Properties

Client	Lemma Finance
Title	Smart Contract Audit Report
Target	Lemma Stablecoin
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 8, 2021	Xiaotao Wu	Final Release
1.0-rc	October 7, 2021	Xiaotao Wu	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 source code of the `Lemma Stablecoin` 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 Lemma Stablecoin

`Lemma Stablecoin`, i.e., `USDL`, is a USD-pegged stable coin that is decentralized, capital efficient (depositing 1 USD of ETH returns 1 `USDL`), and yield-bearing when staked. Users of the protocol are able to mint `USDL` with cryptocurrencies such as ETH and redeem `USDL` for 1 USD worth of cryptocurrency at all times on `Lemma` in a permissionless and non-custodial manner.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Lemma Stablecoin

Item	Description
Name	Lemma Finance
Website	https://lemma.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 8, 2021

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

- https://github.com/lemma-finance/basis-trading-stablecoin.git (4e1361c)

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

- https://github.com/lemma-finance/basis-trading-stablecoin.git (80f1486)

1.2 About PeckShield

PeckShield Inc. [13] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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 accordingly classified into four categories, 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

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

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) [11], 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
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 design and implementation of the `Lemma Stablecoin` protocol smart contracts. 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	2	■ ■
Low	3	■ ■ ■
Informational	1	■
Undetermined	0	
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 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risks In USDLemma::depositTo()	Time and State	Fixed
PVE-002	Low	Improved Handling Of Corner Cases In USDLemma::depositTo()	Business Logic	Fixed
PVE-003	Medium	Possible Costly xUSDL From Improper Pool Initialization	Time and State	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-006	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	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 Potential Reentrancy Risks In USDLemma::depositTo()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `USDLemma`
- Category: Time and State [10]
- CWE subcategory: CWE-841 [6]

Description

While reviewing the current `Lemma Stablecoin` contracts, we notice there is a potential reentrancy risk in the `depositTo()` function of the `USDLemma` contract. To elaborate, we show below the code snippet of this routine.

```
    /// @notice Deposit collateral like WETH, WBTC, etc. to mint USDL
    /// @param to Receipent of minted USDL
    /// @param amount Amount of USDL to mint
    /// @param perpetualDEXIndex Index of perpetual dex, where position will be opened
    /// @param maxCollateralRequired Maximum amount of collateral to be used to mint
        given USDL
    /// @param collateral Collateral to be used to mint USDL
    function depositTo(
        address to,
        uint256 amount,
        uint256 perpetualDEXIndex,
        uint256 maxCollateralRequired,
        IERC20Upgradeable collateral
    ) public {
        IPerpetualDEXWrapper perpDEXWrapper = IPerpetualDEXWrapper(
            perpetualDEXWrappers[perpetualDEXIndex][address(collateral)]
        );
        uint256 collateralRequired = perpDEXWrapper.
            getCollateralAmountGivenUnderlyingAssetAmount(amount, true);
        collateralRequired = perpDEXWrapper.getAmountInCollateralDecimals(
            collateralRequired, true);
```

```
85         require(collateralRequired <= maxCollateralRequired, "collateral required
               execeeds maximum");
86         SafeERC20Upgradeable.safeTransferFrom(collateral, _msgSender(), address(
               perpDEXWrapper), collateralRequired);
87         perpDEXWrapper.open(amount);
88         _mint(to, amount);
89     }
```

Listing 3.1: `USDLemma::depositTo()`

In the `depositTo()` function, we notice `SafeERC20Upgradeable.safeTransferFrom()` (line 86) will be called to transfer collateral tokens from `_msgSender()` to the `perpDEXWrapper` contract. If the collateral token faithfully implements the ERC777-like standard, then the `depositTo()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend` and `tokensReceived` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In the ERC777 token case, the above hook can be planted in `SafeERC20Upgradeable.safeTransferFrom()` (line 86) before the actual transfer of the underlying token occurs. In this particular case, if the external contract has certain hidden logic, we may run into risk of having a `re-entrancy` via other public methods. Note this issue is also present in the `withdrawTo()` routine of the same contract.

Recommendation Add necessary reentrancy guards (e.g., `nonReentrant`) to prevent unwanted reentrancy risks.

Status This issue has been fixed in the following commit: `a213335`.

3.2 Improved Handling Of Corner Cases In USDLemma::depositTo()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: USDLemma
- Category: Business Logic [10]
- CWE subcategory: CWE-837 [5]

Description

The USDLemma contract of Lemma Stablecoin protocol provides a public depositTo() function for users to supply collateral assets to the protocol and and mint the corresponding amount of USDL tokens to the users. While examining the routine, we notice the current implementation can be improved.

To elaborate, we show below its code snippet. When this routine is called by a user, the user needs to specify the perpetualDEXIndex and the collateral (lines 76 and 78). These two parameters determine which perpetual DEX Wrapper contract to use for this user (lines 80-81). If these two parameters are not set correctly by the user, the obtained perpDEXWrapper address might be equal to address(0).

```
67     /// @notice Deposit collateral like WETH, WBTC, etc. to mint USDL
68     /// @param to Receipent of minted USDL
69     /// @param amount Amount of USDL to mint
70     /// @param perpetualDEXIndex Index of perpetual dex, where position will be opened
71     /// @param maxCollateralRequired Maximum amount of collateral to be used to mint
           given USDL
72     /// @param collateral Collateral to be used to mint USDL
73     function depositTo(
74         address to,
75         uint256 amount,
76         uint256 perpetualDEXIndex,
77         uint256 maxCollateralRequired,
78         IERC20Upgradeable collateral
79     ) public {
80         IPerpetualDEXWrapper perpDEXWrapper = IPerpetualDEXWrapper(
81             perpetualDEXWrappers[perpetualDEXIndex][address(collateral)]
82         );
83         uint256 collateralRequired = perpDEXWrapper.
               getCollateralAmountGivenUnderlyingAssetAmount(amount, true);
84         collateralRequired = perpDEXWrapper.getAmountInCollateralDecimals(
               collateralRequired, true);
85         require(collateralRequired <= maxCollateralRequired, "collateral required
               execeeds maximum");
86         SafeERC20Upgradeable.safeTransferFrom(collateral, _msgSender(), address(
               perpDEXWrapper), collateralRequired);
87         perpDEXWrapper.open(amount);
```

```
        _mint(to, amount);
    }
```

Listing 3.2: `USDLemma::depositTo()`

Note this issue is also present in the `withdrawTo()` and `reBalance()` routines of the same contract.

Recommendation Take into consideration the scenario where the obtained `perpDEXWrapper` address might be equal to `address(0)`.

Status This issue has been fixed in the following commit: `a213335`.

3.3 Possible Costly xUSDL From Improper Pool Initialization

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `xUSDL`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [3]

Description

The `xUSDL` contract of `Lemma Stablecoin` protocol provides an external `deposit()` function for users to deposit the `USDL` to the pool and mint the corresponding shares of `xUSDL` tokens to the users. While examining the `xUSDL` token share calculation with the given `USDL` amount, we notice an issue that may unnecessarily make the `xUSDL` token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    /// @notice Deposit and mint xUSDL in exchange of USDL
    /// @param amount of USDL to deposit
    /// @return shares Amount of xUSDL minted
    function deposit(uint256 amount) external override returns (uint256 shares) {
        if (totalSupply() == 0) {
            shares = amount;
        } else {
            shares = (amount * 1e18) / pricePerShare();
        }
        SafeERC20Upgradeable.safeTransferFrom(usdl, _msgSender(), address(this), amount)
            ;
        userUnlockBlock[_msgSender()] = block.number + MINIMUM_LOCK;
        _mint(_msgSender(), shares);
    }
```

Listing 3.3: `xUSDL::deposit()`

Specifically, when the pool is being initialized, the `shares` value directly takes the value of `amount` (line 50), which is manipulatable by the malicious actor. As this is the first provide, the `totalSupply()` equals the `amount = 1 WEI`. With that, the actor can further transfer a huge amount of `USDL` to `xUSDL` contract with the goal of making the `xUSDL` extremely expensive (line 72).

```
69     /// @notice Price per share in terms of USDL
70     /// @return price Price of 1 xUSDL in terms of USDL
71     function pricePerShare() public view override returns (uint256 price) {
72         price = (balance() * 1e18) / totalSupply();
73     }
```

Listing 3.4: `xUSDL::pricePerShare()`

An extremely expensive `xUSDL` 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 `shares` for deposited assets (line 52). If truncated to be zero, the deposited assets are essentially considered dust and kept by the contract without returning any `xUSDL` tokens.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the mint amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the `Lemma Stablecoin` protocol, there is a certain privileged account, i.e., `_owner`. When examining the related contracts, we notice inherent trust on this privileged account. To elaborate, we show below the related functions.

Firstly, a number of `setters`, e.g., `setStakingContractAddress()`, `setLemmaTreasury()`, `setFees()` and `addPerpetualDEXWrapper()`, allow for the `_owner` to set various protocol-wide risk parameters, including `stakingContractAddress`, `lemmaTreasury`, `fees`, and `perpetualDEXWrappers`.

```
    /// @notice Set staking contract address, can only be called by owner
    /// @param _stakingContractAddress Address of staking contract
```

```
    function setStakingContractAddress(address _stakingContractAddress) external onlyOwner {
        stakingContractAddress = _stakingContractAddress;
    }

    /// @notice Set Lemma treasury, can only be called by owner
    /// @param _lemmaTreasury Address of Lemma Treasury
    function setLemmaTreasury(address _lemmaTreasury) external onlyOwner {
        lemmaTreasury = _lemmaTreasury;
    }

    /// @notice Set Fees, can only be called by owner
    /// @param _fees Fees taken by the protocol
    function setFees(uint256 _fees) external onlyOwner {
        fees = _fees;
    }

    /// @notice Add address for perpetual dex wrapper for perpetual index and collateral, can only be called by owner
    /// @param perpetualDEXIndex, index of perpetual dex
    /// @param collateralAddress, address of collateral to be used in the dex
    /// @param perpetualDEXWrapperAddress, address of perpetual dex wrapper
    function addPerpetualDEXWrapper(
        uint256 perpetualDEXIndex,
        address collateralAddress,
        address perpetualDEXWrapperAddress
    ) public onlyOwner {
        perpetualDEXWrappers[perpetualDEXIndex][collateralAddress] = perpetualDEXWrapperAddress;
    }
```

Listing 3.5: `USDLemma::setStakingContractAddress()/setLemmaTreasury()/setFees()/addPerpetualDEXWrapper()`

Secondly, another set of `setters`, i.e., `setUSDLemma()`, `setReferrer()`, `setReBalancer()` and `setMaxPosition()`, allow for the `_owner` to set other risk parameters in `MCDEXLemma`, including `usdLemma`, `referrer`, `reBalancer` and `maxPosition`. Note only the `reBalancer` is allowed to call the `reBalance()` function of the `MCDEXLemma` contract.

```
    ///@notice sets USDLemma address - only owner can set
    ///@param _usdlemma USDLemma address to set
    function setUSDLemma(address _usdlemma) public onlyOwner {
        usdLemma = _usdlemma;
    }

    ///@notice sets referer address - only owner can set
    ///@param _referrer referer address to set
    function setReferrer(address _referrer) external onlyOwner {
        referrer = _referrer;
    }

    ///@notice sets reBalancer address - only owner can set
    ///@param _reBalancer reBalancer address to set
```

```
    function setReBalancer(address _reBalancer) public onlyOwner {
        reBalancer = _reBalancer;
    }

    ///@notice sets Max Positions - only owner can set
    ///@param _maxPosition reBalancer address to set
    function setMaxPosition(uint256 _maxPosition) public onlyOwner {
        maxPosition = _maxPosition;
    }
```

Listing 3.6: `MCDEXLemma::setUSDLemma()/setReferrer()/setReBalancer()/setMaxPosition()`

```
    /// @notice Rebalance position of dex based on accumulated funding, since last rebalancing
    /// @param _reBalancer Address of rebalancer who called function on USDL contract
    /// @param amount Amount of accumulated funding fees used to rebalance by opening or closing a short position
    /// @param data Abi encoded data to call respective mcdex function, contains limitPrice and deadline
    /// @return True if successful, False if unsuccessful
    function reBalance(
        address _reBalancer,
        int256 amount,
        bytes calldata data
    ) external override returns (bool) {
        liquidityPool.forceToSyncState();
        require(_msgSender() == usdLemma, "only usdLemma is allowed");
        require(_reBalancer == reBalancer, "only rebalancer is allowed");

        (int256 limitPrice, uint256 deadline) = abi.decode(data, (int256, uint256));
        int256 fundingPNL = getFundingPNL();

        (int256 tradePrice, int256 totalFee, ) = liquidityPool.queryTrade(
            perpetualIndex,
            address(this),
            amount,
            referrer,
            0
        );
        int256 deltaCash = amount.abs().wmul(tradePrice);
        uint256 collateralAmount = (deltaCash + totalFee).toUint256();
        if (amount < 0) {
            realizedFundingPNL -= collateralAmount.toInt256();
        } else {
            realizedFundingPNL += collateralAmount.toInt256();
        }

        int256 difference = fundingPNL - realizedFundingPNL;
        //error +-10**12 is allowed in calculation
        require(difference.abs() <= 10**12, "not allowed");

        liquidityPool.trade(perpetualIndex, address(this), amount, limitPrice, deadline, referrer, 0);
```

```
        return true;
    }
```

Listing 3.7: MCDEXLemma::reBalance()

Lastly, the updateLock() function allows for the _owner to update MINIMUM_LOCK for the xUSDL contract. If the MINIMUM_LOCK is set too large, the users may not be able to withdraw their deposited assets from the xUSDL contract.

```
    /// @notice updated minimum number of blocks to be locked before xUSDL tokens are
        unlocked
    function updateLock(uint256 lock) external onlyOwner {
        MINIMUM_LOCK = lock;
    }
```

Listing 3.8: xUSDL::updateLock()

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the _owner may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to _owner explicit to Lemma Stablecoin users.

Status This issue has been confirmed. The team further clarifies that they will progressively decentralized where at first it will be a multisig and then once they launch the token and it is sufficiently distributed they will give the administrative privileges to the governance contract. As for re-balancer, it is going to be an EOA at first. The team is working on a smart contract which can handle re-balancer role without depending on a specific EOA.

3.5 Meaningful Events For Important State Changes

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple contracts
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `USDLemma` contract as an example. While examining the events that reflect the `USDLemma` dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the `depositTo()` and `withdrawTo()` functions are being called, there are no corresponding events being emitted to reflect the occurrence of `deposit` and `withdraw` (lines 73 and 97).

```
67      /// @notice Deposit collateral like WETH, WBTC, etc. to mint USDL
68      /// @param to Receipent of minted USDL
69      /// @param amount Amount of USDL to mint
70      /// @param perpetualDEXIndex Index of perpetual dex, where position will be opened
71      /// @param maxCollateralRequired Maximum amount of collateral to be used to mint given USDL
72      /// @param collateral Collateral to be used to mint USDL
73      function depositTo(
74          address to,
75          uint256 amount,
76          uint256 perpetualDEXIndex,
77          uint256 maxCollateralRequired,
78          IERC20Upgradeable collateral
79      ) public {
80          IPerpetualDEXWrapper perpDEXWrapper = IPerpetualDEXWrapper(
81              perpetualDEXWrappers[perpetualDEXIndex][address(collateral)]
82          );
83          uint256 collateralRequired = perpDEXWrapper.getCollateralAmountGivenUnderlyingAssetAmount(amount, true);
84          collateralRequired = perpDEXWrapper.getAmountInCollateralDecimals(collateralRequired, true);
85          require(collateralRequired <= maxCollateralRequired, "collateral required execeeds maximum");
86          SafeERC20Upgradeable.safeTransferFrom(collateral, _msgSender(), address(perpDEXWrapper), collateralRequired);
87          perpDEXWrapper.open(amount);
88          _mint(to, amount);
89      }
90
91      /// @notice Redeem USDL and withdraw collateral like WETH, WBTC, etc
92      /// @param to Receipent of withdrawn collateral
93      /// @param amount Amount of USDL to redeem
94      /// @param perpetualDEXIndex Index of perpetual dex, where position will be closed
95      /// @param minCollateralToGetBack Minimum amount of collateral to get back on redeeming given USDL
96      /// @param collateral Collateral to be used to redeem USDL
97      function withdrawTo(
98          address to,
99          uint256 amount,
```

```
            uint256 perpetualDEXIndex,
            uint256 minCollateralToGetBack,
            IERC20Upgradeable collateral
        ) public {
            _burn(_msgSender(), amount);
            IPerpetualDEXWrapper perpDEXWrapper = IPerpetualDEXWrapper(
                perpetualDEXWrappers[perpetualDEXIndex][address(collateral)]
            );
            uint256 collateralToGetBack = perpDEXWrapper.
                getCollateralAmountGivenUnderlyingAssetAmount(amount, false);
            collateralToGetBack = perpDEXWrapper.getAmountInCollateralDecimals(
                collateralToGetBack, false);
            require(collateralToGetBack >= minCollateralToGetBack, "collateral got back
                is too low");
            perpDEXWrapper.close(amount);
            SafeERC20Upgradeable.safeTransfer(collateral, to, collateralToGetBack);
        }
```

Listing 3.9: `USDLemma::depositTo()/withdrawTo()`

Note a number of routines in the `Lemma Stablecoin` contracts can be similarly improved, including `USDLemma::reBalance()/setStakingContractAddress()/setLemmaTreasury()/setFees()/addPerpetualDEXWrapper()`, `MCDEXLemma::setUSDLemma()/setReferrer()/setReBalancer()/setMaxPosition()`, `xUSDL::updateLock()`, and `xUSDL::deposit()/withdraw()`.

Recommendation Properly emit the related events when the above-mentioned functions are being called.

Status This issue has been fixed in the following commit: `2dcb1bc`.

3.6 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple contracts`
- Category: Coding Practices [9]
- 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 analyze 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).

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

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.10: USDT Token **Contract**

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.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20Upgradeable token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
```

```
    }
```

Listing 3.11: `SafeERC20Upgradeable::safeApprove()`

In the following, we show the `resetApprovals()` routine in the `MCDEXLemma` contract. Although `safeApprove()` is used, the calling of this routine may still revert as it requires `(value == 0)|| (token.allowance(address(this), spender)== 0)` in the `SafeERC20Upgradeable` contract's `safeApprove()` implementation.

```
    /// @notice reset approvals
    function resetApprovals() external {
        SafeERC20Upgradeable.safeApprove(collateral, address(liquidityPool), MAX_UINT256
            );
    }
```

Listing 3.12: `MCDEXLemma::resetApprovals()`

Note the `resetApprovals()` routine in the `xUSDL` contract can be similarly improved.

Recommendation Reducing the allowance to 0 first when resetting the approvals. An example revision is shown below.

```
    /// @notice reset approvals
    function resetApprovals() external {
        SafeERC20Upgradeable.safeApprove(collateral, address(liquidityPool), 0);
        SafeERC20Upgradeable.safeApprove(collateral, address(liquidityPool), MAX_UINT256
            );
    }
```

Listing 3.13: `MCDEXLemma::resetApprovals()`

Status The issue has been confirmed.

4 | Conclusion

In this audit, we have analyzed the Lemma Stablecoin design and implementation. Lemma Stablecoin is a USD-pegged stable coin that is decentralized, capital efficient, and yield-bearing when staked. The current code base is well structured and neatly organized. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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