ENTERSOFT

Cicero

Smart Contract Audit

DATE: 1st August 2024 | PREPARED BY: Entersoft Australia

Contents

Revision History & Version Control

Start Date	End Date	Author	Comments/Details
30 Jun 2024	28 Jun 2024	Gurkirat	Interim Release for the Client

Reviewed by	Released by
Nishita Palaksha	Nishita Palaksha

Entersoft was commissioned to perform a source code review on Cicero smart contracts. The review was conducted between May 30, 2024, to June 28, 2024. The report is organized into the following sections.

- Executive Summary: A high-level overview of the security audit findings.
- Technical analysis: Our detailed analysis of the Smart Contract code

The information in this report should be used to understand overall code quality, security, correctness, and meaning that code will work as described in the smart contract.

ENTERSOFT

1.0 Disclaimer

This is a limited audit report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to: (i) smart contract best coding practices and vulnerabilities in the framework and algorithms based on white paper, code, the details of which are set out in this report, (Smart Contract audit). To get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us based on what it says or does not say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Entersoft Australia and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Entersoft) owe no duty of care towards you or any other person, nor does Entersoft make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Entersoft hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Entersoft hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Entersoft, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the Smart contract is purely based on the smart contract code shared with us alone.

2.0 Overview

2.1 Project Overview

During the period of **30 Jun 2024 to 28 Jun 2024**, Entersoft performed smart contract security audits for **Cicero**.

2.2 Scope

The scope of this audit was to analyze and document the smart contract codebase for quality, security, and correctness.

The following files were reviewed as part of the scope:

- IRealMath.sol
- LoanFactory.sol
- IToken.sol
- LodaAcl.sol
- Atticus.sol
- USDC.sol
- SLPToken.sol
- CiceroToken.sol
- CiceroNFT.sol
- LodaSLP.sol
- LodaJLP.sol
- LodaILP.sol
- LoanCommon.sol
- StakeCommons.sol
- ILPCommon.sol
- RealMath.sol
- Loan.sol
- CommonCompute.sol
- ILPCompute.sol
- JLPCompute.sol
- LodaCompute.sol
- SLPCompute.sol

Commit: 502c4e70b86e6be50ee0bb2a205a07dd662afe59

OUT-OF-SCOPE: External contracts, External Oracles, other smart contracts in the repository, or imported smart contracts.

2.3 Project Summary

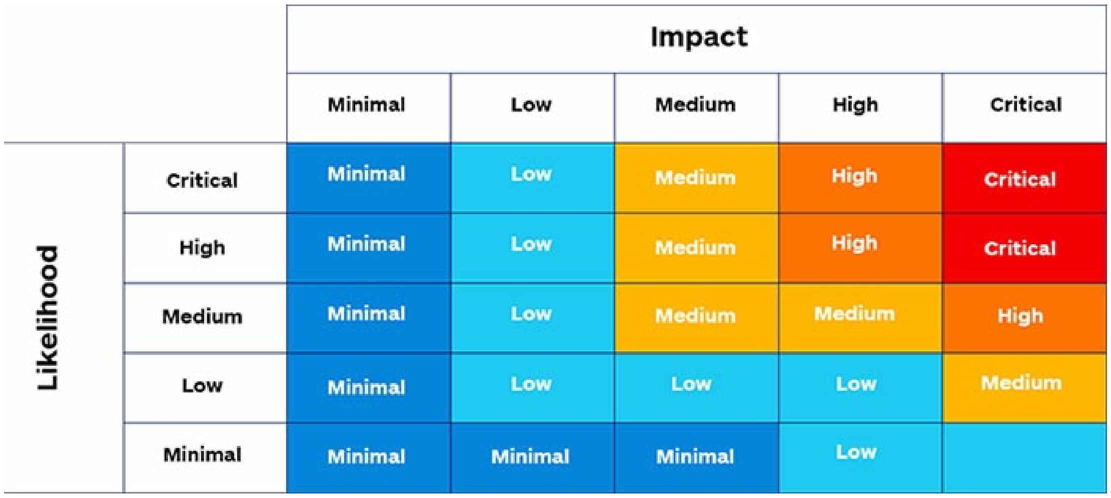
Project Name	No. of Smart Contract File(s)	Verified	Vulnerabilities
Cicero	22	Yes	As per report. Section 2.6

2.4 Audit Summary

Delivery Date	Method of Audit	Consultants Engaged
01 Aug 2024	Manual and Automated approach	3

2.5 Security Level References

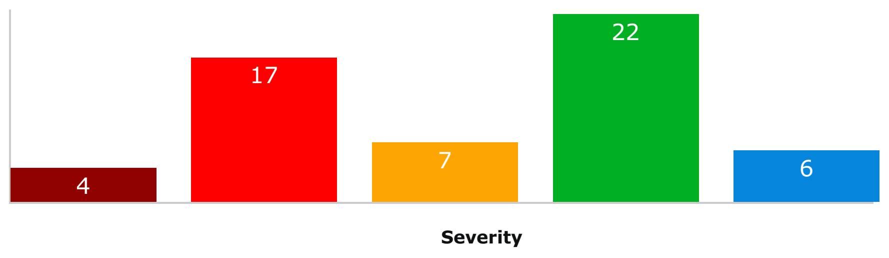
Every vulnerability in this report was assigned a severity level from the following classification table:



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

2.6 Vulnerability Summary

● Critical	● High	● Medium	● Low	● Informational
4	17	7	22	6


4
17
7
22
6
Severity

3.0 Executive Summary

Entersoft has conducted a comprehensive technical audit of the Cicero protocol through a comprehensive smart contract audit approach. The primary objective was to identify potential vulnerabilities and security risks within the codebase, ensuring adherence to industry-leading standards while prioritizing security, reliability, and performance. Our focus was on prompt and efficient identification and resolution of vulnerabilities to enhance the overall robustness of the solidity smart contract.

Importantly, our audit process intentionally avoided reliance solely on automated tools, emphasizing a more in-depth and nuanced approach to security analysis. Conducted from May 30, 2024, to June 28, 2024, our team diligently assessed and validated the security posture of the solidity smart contract, ultimately finding a number of vulnerabilities as per vulnerability summary table.

Testing Methodology:
We have leveraged static analysis techniques extensively to identify potential vulnerabilities automatically with the aid of cutting-edge tools such as Slither and Aderyn. Apart from this, we carried out extensive manual testing to iron out vulnerabilities that could slip through an automated check. This included a variety of attack vectors like reentrancy attacks, overflow and underflow attacks, timestamp dependency attacks, and more.
While going through the due course of this audit, we also ensured to cover edge cases, and built a combination of scenarios to assess the contracts' resilience. Our attempt to leave no stone unturned involved coming up with both negative and positive test cases for the system, and grace handling of stressed scenarios.
Our testing methodology in Solidity adhered to industry standards and best practices, integrating partially implemented OWASP and NIST SP 800 standards for encryption and signatures. Solidity's renowned security practices were complemented by tools such as Solhint for linting, and the Solidity compiler for code optimization. Sol-profiler, Sol-coverage, and Sol-sec were employed to ensure code readability and eliminate unnecessary dependencies.

Tools Used for Audit:
During our audit, we utilized a suite of tools to enhance the security and performance of our program. Our team combined their expertise and industry best practices with tools integrated into our development environment, such as Slither and Aderyn. This comprehensive approach ensures a thorough analysis, identifying potential issues that automated tools alone might miss. At Entersoft, we take pride in using these tools, which greatly contribute to the quality, security, and maintainability of our codebase.

Code Review / Manual Analysis:
Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analyzed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and

ensure that the smart contract was secure and reliable.

Auditing Approach and Methodologies Applied:
The solidity smart contract was audited in a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of code:

- Code quality and structure: We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analyzing the overall architecture of the solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.

- Security vulnerabilities: Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, Signatures, and deprecated functions.

3.1 Findings

Vulnerability ID	Contract Name	Severity	Status
1	LodaILP.sol	● Critical	Pending
2	loan.sol	● Critical	Pending
3	Loan.sol, LodaJLP.sol, LodaILP.sol, LodaSLP.sol	● Critical	Pending
4	loanFactory.sol	● Critical	Pending
5	LoanFactory.sol, ILPCompute.sol, LodaILP.sol, MockedLodaSLP.sol, USDCToken.sol, Atticus.sol, CiceroNFT.sol, CiceroToken.sol, SLPToken.sol,	● High	Pending
6	loanFactory.sol	● High	Pending
7	loanFactory.sol	● High	Pending
8	LodaSLP.sol	● High	Pending
9	loan.sol	● High	Pending
10	LodaILP.sol	● High	Pending
11	LodaJLP.sol	● High	Pending

12	LodaSLP.sol	● High	Pending
13	LodaJLP.sol	● High	Pending
14	LodaSLP.sol	● High	Pending
15	LodaILP.sol	● High	Pending
16	LodaJLP.sol	● High	Pending
17	LodaSLP.sol	● High	Pending
18	LodaSLP.sol	● High	Pending
19	LodaACL.sol	● High	Pending
20	LodaJLP.sol	● High	Pending
21	LodaILP.sol	● High	Pending
22	LodaSLP.sol	● Medium	Pending
23	LodaCompute.sol, Loan.sol, LodaJLP.sol	● Medium	Pending
24	LodaSLP.sol	● Medium	Pending
25	LodaJLP.sol	● Medium	Pending
26	LodaILP.sol	● Medium	Pending
27	LodaJLP.sol	● Medium	Pending
28	LodaSLP.sol	● Medium	Pending
29	LodaJLP.sol	● Low	Pending
30	LodaJLP.sol	● Low	Pending
31	LodaSLP.sol	● Low	Pending
32	LodaILP.sol	● Low	Pending
33	LodaSLP.sol	● Low	Pending
34	LodaILP.sol	● Low	Pending
35	LodaILP.sol	● Low	Pending
36	LodaSLP.sol	● Low	Pending

37	LodaSLP.sol	● Low	Pending
38	LodaSLP.sol	● Low	Pending
39	LodaJLP.sol	● Low	Pending
40	loan.sol	● Low	Pending
41	LodaSLP.sol	● Low	Pending
42	LodaJLP.sol	● Low	Pending
43	LodaILP.sol	● Low	Pending
44	loan.sol	● Low	Pending
45	LodaSLP.sol, CiceroNFT.sol	● Low	Pending
46	LoanFactory.sol, ILPCompute.sol, Loan.sol, LodaILP.sol, LodaJLP.sol, LodaSLP.sol, CiceroNFT.sol	● Low	Pending
47	LodaSLP.sol	● Low	Pending
48	LodaJLP.sol	● Low	Pending
49	loanFactory.sol	● Low	Pending
50	LoanFactory.sol	● Low	Pending
51	LodaILP.sol	● Informational	Pending
52	LodaJLP.sol	● Informational	Pending
53	LodaSLP.sol	● Informational	Pending
54	LoanFactory.sol	● Informational	Pending
55	Loan.sol, LodaJLP.sol, LodaILP.sol, LodaSLP.sol	● Informational	Pending
56	LodaSLP.sol	● Informational	Pending

3.2 Recommendations

The smart contracts need further refinement to fully align with best security practices and industry standards.

Our analysis has identified several vulnerabilities, including access control issues, reentrancy concerns, and business logic errors. We recommend addressing these issues before considering deployment on the mainnet or any production environment to ensure optimal security and functionality.

4.0 Technical Analysis

4.1 Correct implementation required for withdrawInit Function

Severity	**Status**	**Type of Analysis**
● Critical	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The withdrawInit function is designed to update the coolOffStarted timestamp for a given stake index. This function is critical as it initiates the cool-off period for the withdrawal process, allowing users to start the process of withdrawing their staked assets after a specified period.

Locations:

function withdrawInit: Line no = 492-495

function suffrage: Line no = 538 - 562

function withdraw: Line no = 249- 281

function canClaimRewards: Line no = 365-381

function rewardsAccruedPerStake: Line no = 387 - 427

Remediation:

- To mitigate this vulnerability, ensure the withdrawInit function is implemented correctly which means it define the purpose correctly if it should be called after or under withdraw or staking function or any other function as per the requirement.
- Restrict access to the function using access modifiers (private or internal) or implement role-based access controls to ensure only authorized contracts or roles can update profit information.
- After correctly implementing the function, ensure the functions dependent on this variable are working as expected.

Impact:

- The current implementation of the withdrawInit function allows any user to call it, thereby updating the coolOffStarted timestamp for any stake index. This lack of correct implementation for this function makes the smart contract vulnerable to unauthorized manipulation. Since many other functions depend on the cool-off period, this vulnerability can lead to unintended behavior and potential security breaches for the protocol.

Code Snippet:

```
        function withdrawInit(uint256 _stakeIndex) external {

        // claimAccruedRewards(_stakeIndex);

        poolStakes[_stakeIndex].coolOffStarted = block.timestamp;

    }



//Function dependent on coolOffStarted variable are below



function suffrage(uint256 _stakeIndex) public view returns (uint256) {

        if (poolStakes[_stakeIndex].coolOffStarted != COOL_OFF_NONE) return 0;

        uint256 tokensStaked = (poolStakes[_stakeIndex].stakeType ==

            ILPCommon.StakedTokenType.USDC)

            ? usdcToLoda(poolStakes[_stakeIndex].amount)

            : poolStakes[_stakeIndex].amount;

        uint256 tenure = (poolStakes[_stakeIndex].maturityDate -

            poolStakes[_stakeIndex].stakedAt);

        uint256 time = poolStakes[_stakeIndex].stakedAt;

        uint256 multiplyPrecision = 100;

        if (tenure >= (block.timestamp - time)) {

            uint256 daysRemaining = ((((time + tenure) - block.timestamp) *

                multiplyPrecision) / ONE_DAY);

            if (daysRemaining < suffrageDecimation * multiplyPrecision) {

                return

                    maxVotingPower(tokensStaked, daysRemaining) /

                    multiplyPrecision;
            } else {

                return maxVotingPower(tokensStaked);

            }

        } else {

            // voting power will be zero when tenure is expired

            return 0;

        }

    }




function withdraw(uint256 _stakeIndex) external {

        Stake memory _stake = poolStakes[_stakeIndex];


        require(

            _stake.status == StakeCommons.StakeStatus.STAKED,

            "Already Withdrawn"

        );


        require(

            _stake.maturityDate <= block.timestamp ||

                (_stake.coolOffStarted != 0 &&

                    getCoolOffPeriod(_stakeIndex) + _stake.coolOffStarted <=

                    block.timestamp),

            "still within cooling off period"

        );

        IToken lodaToken = IToken(ciceroTokenAddress);

        IToken lodestarToken = IToken(atticusTokenAddress);


        uint256 currentLodaAmount = getLodestarToLoda(_stake.atticusTokens);

        require(
            currentLodaAmount <= lodaToken.balanceOf(address(this)),

            "Insufficient loda tokens in pool"

        );

        lodestarToken.burn(msg.sender, _stake.atticusTokens);

        lodaToken.transfer(msg.sender, currentLodaAmount);

        uint256 claimableRewards = computeRewardsForStake(_stakeIndex);

        if (lodaToken.balanceOf(address(this)) >= claimableRewards) {

            lodaToken.transfer(msg.sender, claimableRewards);

            poolStakes[_stakeIndex].rewardsAlreadyClaimed += claimableRewards;

        }



        poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN;

    }



function canClaimRewards(uint256 _stakeIndex) internal view returns (bool) {

        bool isStaked = poolStakes[_stakeIndex].status ==

            StakeCommons.StakeStatus.STAKED;



        bool isCoolOffStarted = poolStakes[_stakeIndex].coolOffStarted !=

            COOL_OFF_NONE;



        bool isCoolOffNotValid = (!isCoolOffStarted ||

            poolStakes[_stakeIndex].coolOffStarted +

                getCoolOffPeriod(_stakeIndex) >=

            (poolStakes[_stakeIndex].maturityDate - block.timestamp));



        bool isNotMatured = poolStakes[_stakeIndex].maturityDate >=

            block.timestamp;



        return isStaked && isCoolOffNotValid && isNotMatured;
        return isStaked && isCoolOffNotValid && isNotMatured;
    }

function rewardsAccruedPerStake(uint256 _stakeIndex)
        public
        view
        returns (uint256)
    {
        Stake memory stakeData = poolStakes[_stakeIndex];
        if (
            stakeData.coolOffStarted == COOL_OFF_NONE ||
            poolStakes[_stakeIndex].coolOffStarted +
                getCoolOffPeriod(_stakeIndex) >=
            block.timestamp
        ) {
            if (stakeData.maturityDate >= block.timestamp) {
                return
                    stakeData.rewardRatePerSec *
                    (block.timestamp - stakeData.stakedAt) *
                    stakeData.amount;
            } else {
                return
                    stakeData.rewardRatePerSec *
                    (stakeData.maturityDate - stakeData.stakedAt) *
                    stakeData.amount;
            }
        } else {
            if (
                stakeData.coolOffStarted + getCoolOffPeriod(_stakeIndex) >=
                block.timestamp
            ) {
                return
                    stakeData.rewardRatePerSec *
                    (block.timestamp - stakeData.stakedAt) *
                    stakeData.amount;
            } else {
                return
                    stakeData.rewardRatePerSec *
                    ((stakeData.coolOffStarted +
                        getCoolOffPeriod(_stakeIndex)) - stakeData.stakedAt) *
                    stakeData.amount;
            }
        }
    }
```

Reference:

Proof of Vulnerability:

N.A.

4.2 Insecure changeLoanStatus Function

Severity	Status	Type of Analysis
● Critical	Identified	Dynamic

Contract Name:

loan.sol

Description:

The changeLoanStatus function allows changing the status of a loan and triggering related actions. However, it can currently be called by anyone, which introduces a significant security vulnerability.

Locations:

function changeLoanStatus: Line no = 99-112

Remediation:

- Protocol needs to develop a system to check whether this function isn’t being called by malicious user.

Impact:

- Unauthorized Access: Any user can call this function, allowing them to change the loan status, collect loan amounts, and transfer funds.
- If loan is not approved the also any user can call this function and change the status and borrower can claim the loan.

Code Snippet:

```
        function changeLoanStatus(uint8 _status) public {

        require(_status <= uint8(LoanCommon.LoanStatus.FAIL), "Out of range");

        loanData.loanStatus = LoanCommon.LoanStatus(_status);

        if (_status == 1) {

            LoanFactory(loanFactoryAddress).collectLoanAmount(loanData.loanId);

            IToken usdcToken = IToken(usdcTokenAddress);

            usdcToken.transfer(loanData.borrower, loanData.loanAmount);

        }

        if (_status == 2) {

            loanData.maturityDate = block.timestamp + loanData.tenure;

            loanData.loanStartDate = block.timestamp;

            loanData.nextDueDate = block.timestamp + (30 days);

        }

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.3 Reentrancy Vulnerabilities

Severity	Status	Type of Analysis
● Critical	Identified	Static

Contract Name:

Loan.sol, LodaJLP.sol, LodaILP.sol, LodaSLP.sol

Description:

The functions inside smart contracts for Cicero Protocol contain alot of reentrancy vulnerabilities.

Locations:

Loan.sol, LodaJLP.sol, LodaILP.sol, LodaSLP.sol

Remediation:

- To mitigate the reentrancy vulnerabilities, perform state changes after external calls have been made. This ensures that no external calls can interfere with the state changes in progress. Use libraries like openzeppelin reentrancy guard etc.

Impact:

Impact:

- A state variable is changed after a contract uses `call.value`. The attacker uses a fallback function—which is automatically executed after Ether is transferred from the targeted contract—to execute the vulnerable function again, *before* the state variable is changed.

Attack:

- Reentrancy attacks target smart contracts through a malicious exploit. The attacker abuses a smart contract's call again feature to withdraw the budget again and again before the agreement can update its balance.

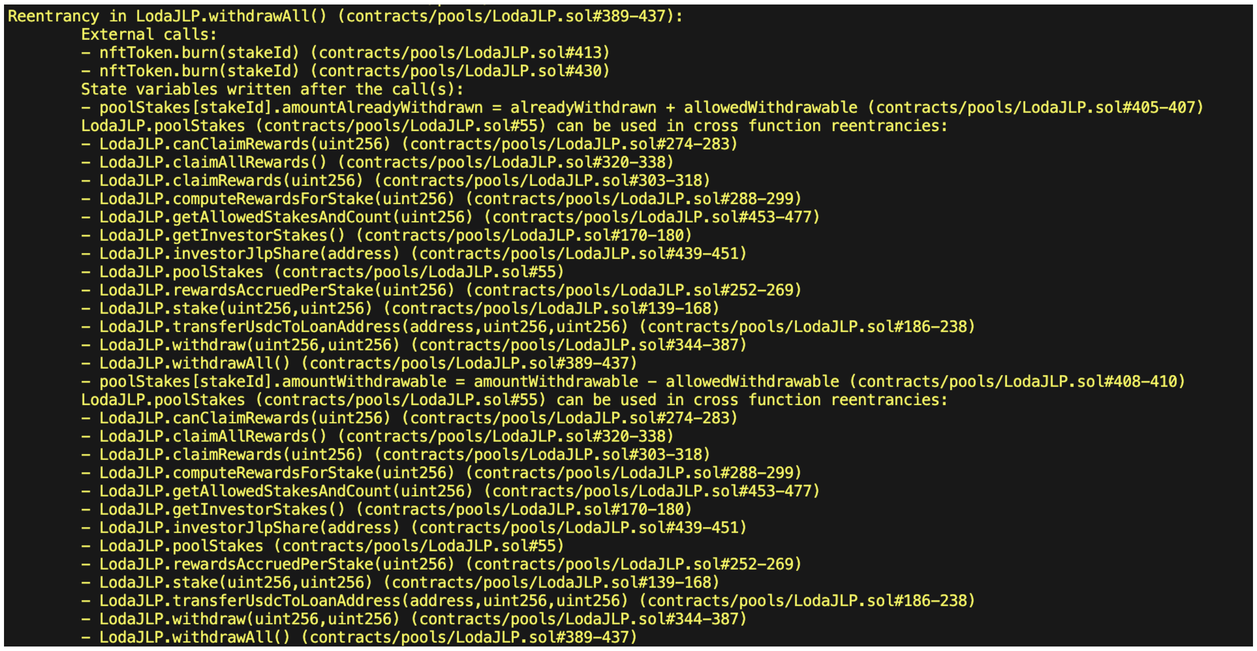
Code Snippet:

NA

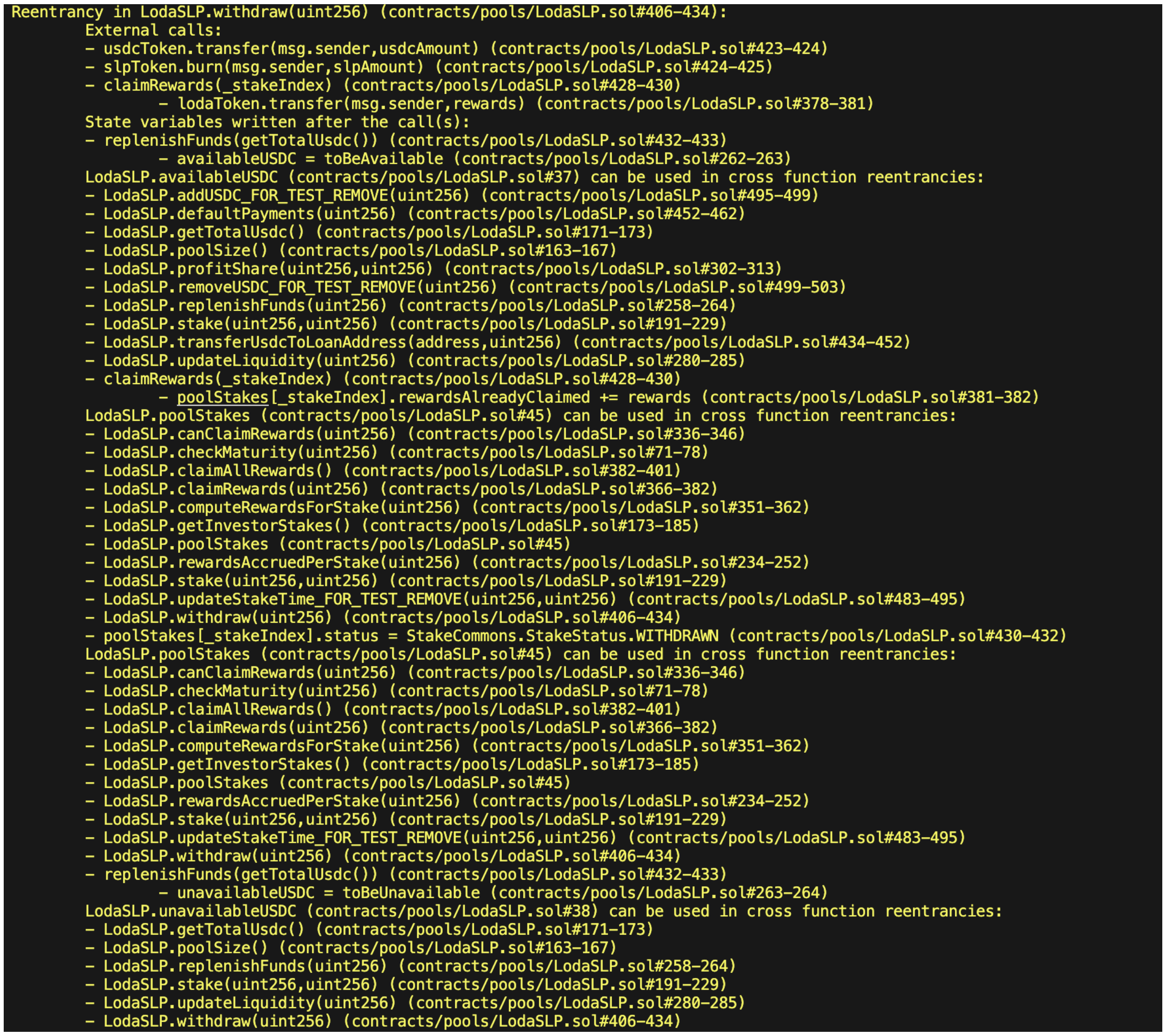
Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1 https://medium.com/@thecoinrepublic01/heres-how-to-prevent-reentrancy-attacks-in-smart-contracts-2b601c1632e7 https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

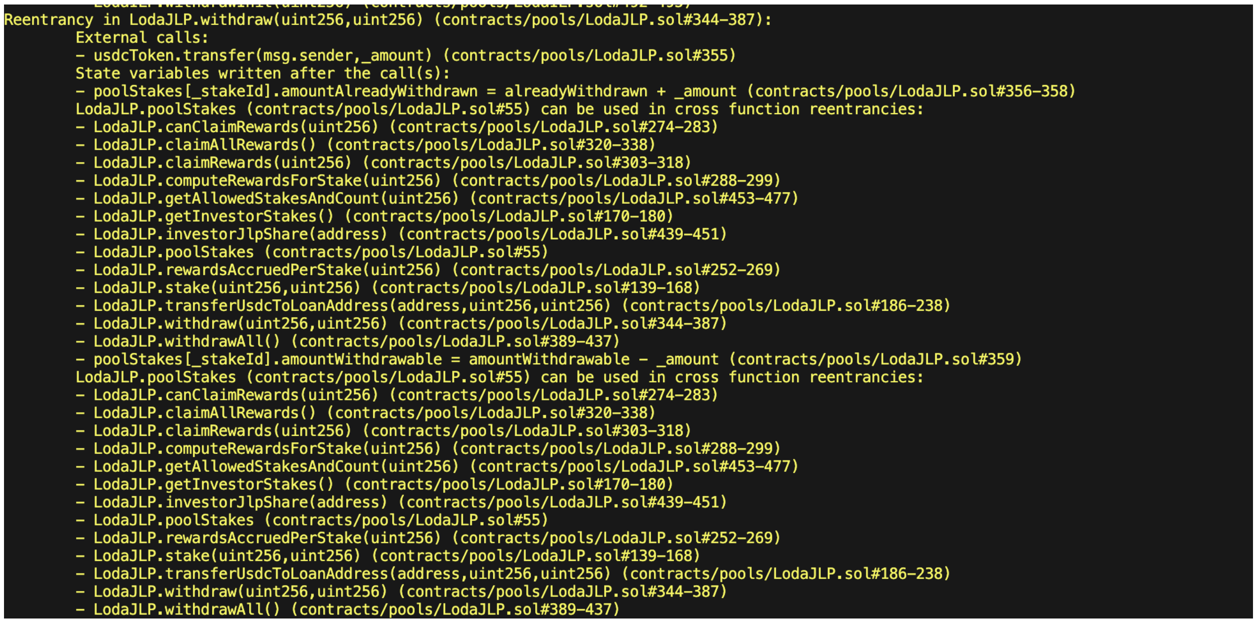
Proof of Vulnerability:



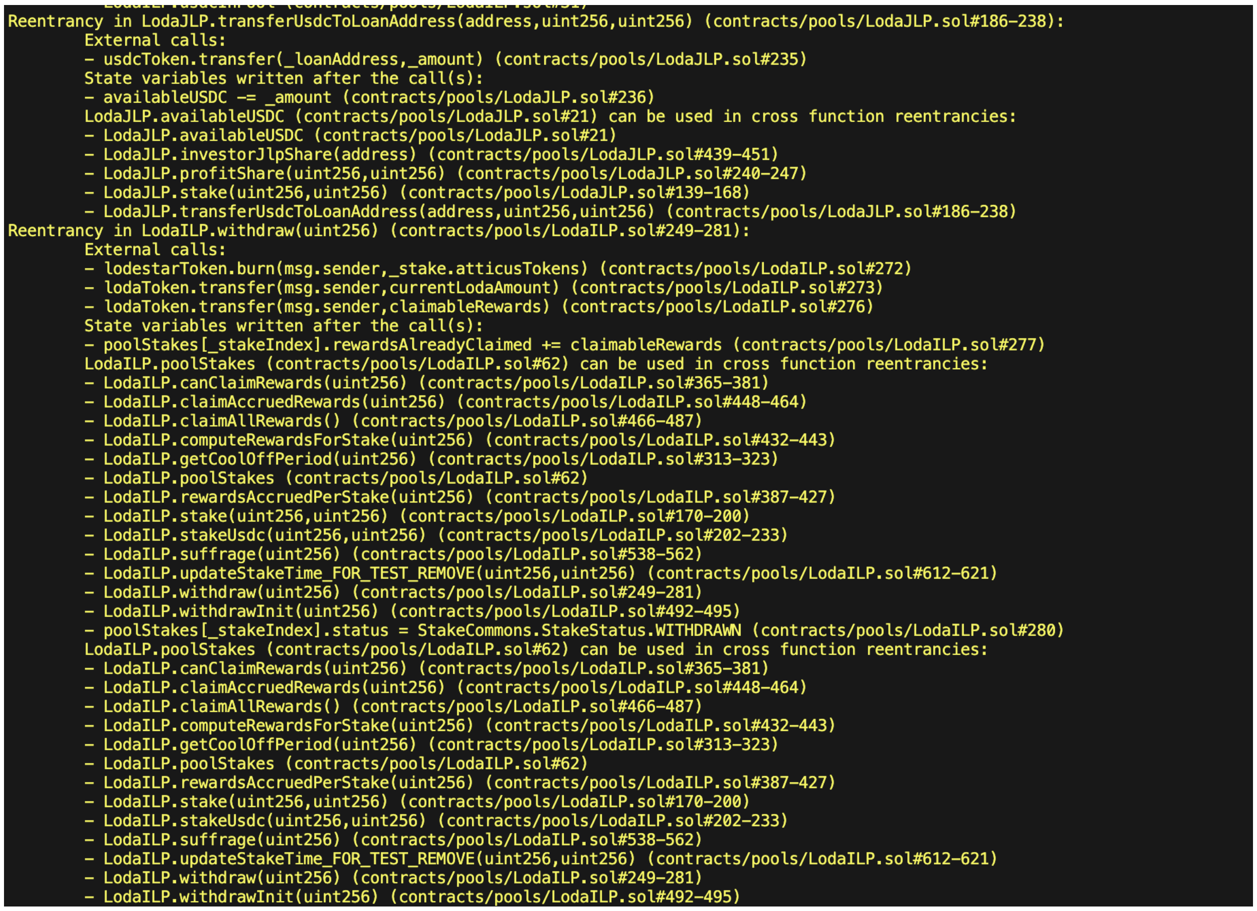
```
Reentrancy in LodaJLP.withdrawAll() (contracts/pools/LodaJLP.sol#389-437):
        External calls:
        - nftToken.burn(stakeId) (contracts/pools/LodaJLP.sol#413)
        - nftToken.burn(stakeId) (contracts/pools/LodaJLP.sol#430)
        State variables written after the call(s):
        - poolStakes[stakeId].amountAlreadyWithdrawn = alreadyWithdrawn + allowedWithdrawable (contracts/pools/LodaJLP.sol#405-407)
        LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55) can be used in cross function reentrancies:
        - LodaJLP.canClaimRewards(uint256) (contracts/pools/LodaJLP.sol#274-283)
        - LodaJLP.claimAllRewards() (contracts/pools/LodaJLP.sol#320-338)
        - LodaJLP.claimRewards(uint256) (contracts/pools/LodaJLP.sol#303-318)
        - LodaJLP.computeRewardsForStake(uint256) (contracts/pools/LodaJLP.sol#288-299)
        - LodaJLP.getAllowedStakesAndCount(uint256) (contracts/pools/LodaJLP.sol#453-477)
        - LodaJLP.getInvestorStakes() (contracts/pools/LodaJLP.sol#170-180)
        - LodaJLP.investorJlpShare(address) (contracts/pools/LodaJLP.sol#439-451)
        - LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55)
        - LodaJLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaJLP.sol#252-269)
        - LodaJLP.stake(uint256,uint256) (contracts/pools/LodaJLP.sol#139-168)
        - LodaJLP.transferUsdcToLoanAddress(address,uint256,uint256) (contracts/pools/LodaJLP.sol#186-238)
        - LodaJLP.withdraw(uint256,uint256) (contracts/pools/LodaJLP.sol#344-387)
        - LodaJLP.withdrawAll() (contracts/pools/LodaJLP.sol#389-437)
        - poolStakes[stakeId].amountWithdrawable = amountWithdrawable - allowedWithdrawable (contracts/pools/LodaJLP.sol#408-410)
        LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55) can be used in cross function reentrancies:
        - LodaJLP.canClaimRewards(uint256) (contracts/pools/LodaJLP.sol#274-283)
        - LodaJLP.claimAllRewards() (contracts/pools/LodaJLP.sol#320-338)
        - LodaJLP.claimRewards(uint256) (contracts/pools/LodaJLP.sol#303-318)
        - LodaJLP.computeRewardsForStake(uint256) (contracts/pools/LodaJLP.sol#288-299)
        - LodaJLP.getAllowedStakesAndCount(uint256) (contracts/pools/LodaJLP.sol#453-477)
        - LodaJLP.getInvestorStakes() (contracts/pools/LodaJLP.sol#170-180)
        - LodaJLP.investorJlpShare(address) (contracts/pools/LodaJLP.sol#439-451)
        - LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55)
        - LodaJLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaJLP.sol#252-269)
        - LodaJLP.stake(uint256,uint256) (contracts/pools/LodaJLP.sol#139-168)
        - LodaJLP.transferUsdcToLoanAddress(address,uint256,uint256) (contracts/pools/LodaJLP.sol#186-238)
        - LodaJLP.withdraw(uint256,uint256) (contracts/pools/LodaJLP.sol#344-387)
        - LodaJLP.withdrawAll() (contracts/pools/LodaJLP.sol#389-437)
```



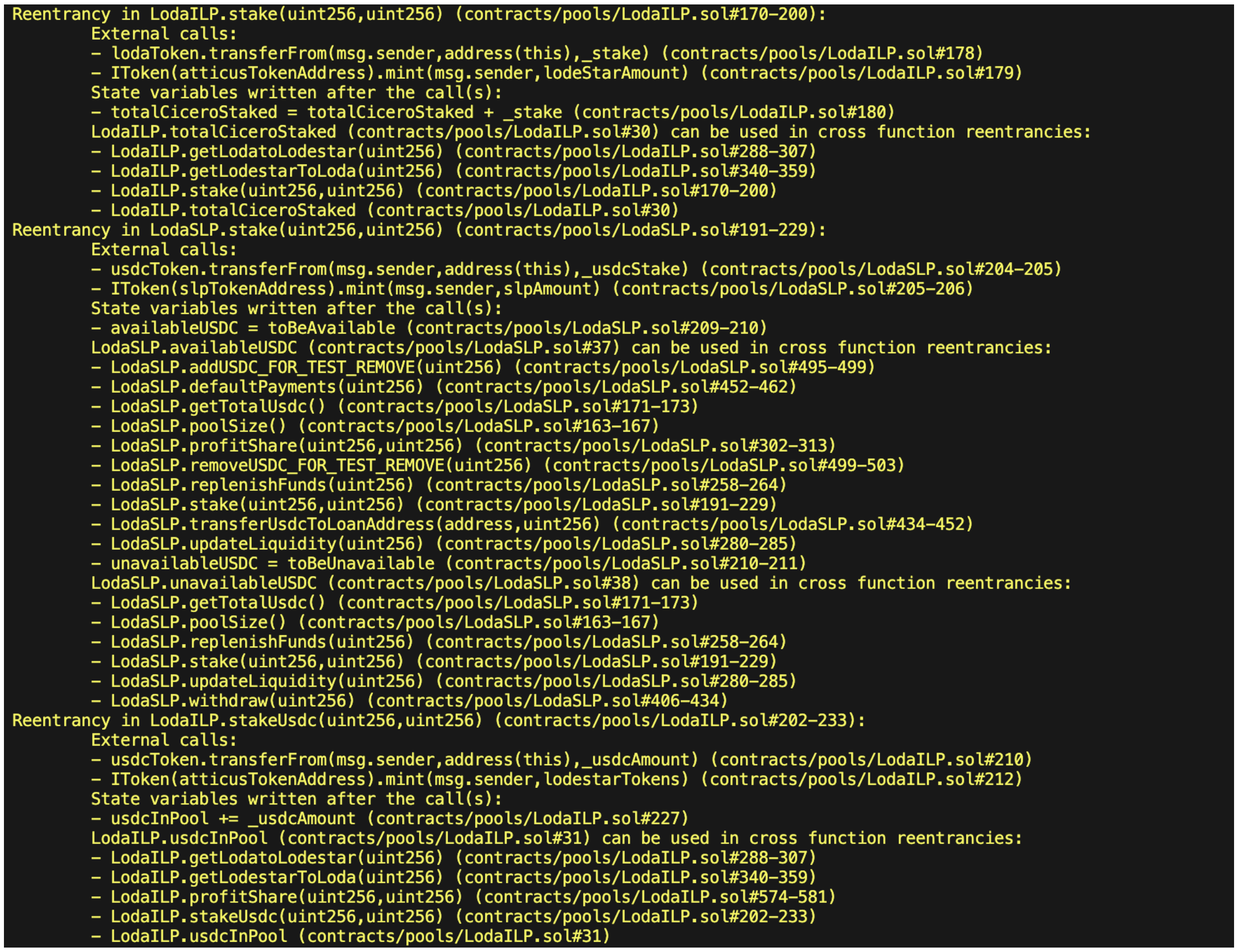
```
Reentrancy in LodaSLP.withdraw(uint256) (contracts/pools/LodaSLP.sol#406-434):
        External calls:
        - usdcToken.transfer(msg.sender,usdcAmount) (contracts/pools/LodaSLP.sol#423-424)
        - slpToken.burn(msg.sender,slpAmount) (contracts/pools/LodaSLP.sol#424-425)
        - claimRewards(_stakeIndex) (contracts/pools/LodaSLP.sol#428-430)
                - lodaToken.transfer(msg.sender,rewards) (contracts/pools/LodaSLP.sol#378-381)
        State variables written after the call(s):
        - replenishFunds(getTotalUsdc()) (contracts/pools/LodaSLP.sol#432-433)
                - availableUSDC = toBeAvailable (contracts/pools/LodaSLP.sol#262-263)
        LodaSLP.availableUSDC (contracts/pools/LodaSLP.sol#37) can be used in cross function reentrancies:
        - LodaSLP.addUSDC_FOR_TEST_REMOVE(uint256) (contracts/pools/LodaSLP.sol#495-499)
        - LodaSLP.defaultPayments(uint256) (contracts/pools/LodaSLP.sol#452-462)
        - LodaSLP.getTotalUsdc() (contracts/pools/LodaSLP.sol#171-173)
        - LodaSLP.poolSize() (contracts/pools/LodaSLP.sol#163-167)
        - LodaSLP.profitShare(uint256,uint256) (contracts/pools/LodaSLP.sol#302-313)
        - LodaSLP.removeUSDC_FOR_TEST_REMOVE(uint256) (contracts/pools/LodaSLP.sol#499-503)
        - LodaSLP.replenishFunds(uint256) (contracts/pools/LodaSLP.sol#258-264)
        - LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229)
        - LodaSLP.transferUsdcToLoanAddress(address,uint256) (contracts/pools/LodaSLP.sol#434-452)
        - LodaSLP.updateLiquidity(uint256) (contracts/pools/LodaSLP.sol#280-285)
        - claimRewards(_stakeIndex) (contracts/pools/LodaSLP.sol#428-430)
                - poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards (contracts/pools/LodaSLP.sol#381-382)
        LodaSLP.poolStakes (contracts/pools/LodaSLP.sol#45) can be used in cross function reentrancies:
        - LodaSLP.canClaimRewards(uint256) (contracts/pools/LodaSLP.sol#336-346)
        - LodaSLP.checkMaturity(uint256) (contracts/pools/LodaSLP.sol#71-78)
        - LodaSLP.claimAllRewards() (contracts/pools/LodaSLP.sol#382-401)
        - LodaSLP.claimRewards(uint256) (contracts/pools/LodaSLP.sol#366-382)
        - LodaSLP.computeRewardsForStake(uint256) (contracts/pools/LodaSLP.sol#351-362)
        - LodaSLP.getInvestorStakes() (contracts/pools/LodaSLP.sol#173-185)
        - LodaSLP.poolStakes (contracts/pools/LodaSLP.sol#45)
        - LodaSLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaSLP.sol#234-252)
        - LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229)
        - LodaSLP.updateStakeTime_FOR_TEST_REMOVE(uint256,uint256) (contracts/pools/LodaSLP.sol#483-495)
        - LodaSLP.withdraw(uint256) (contracts/pools/LodaSLP.sol#406-434)
        - poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN (contracts/pools/LodaSLP.sol#430-432)
        LodaSLP.poolStakes (contracts/pools/LodaSLP.sol#45) can be used in cross function reentrancies:
        - LodaSLP.canClaimRewards(uint256) (contracts/pools/LodaSLP.sol#336-346)
        - LodaSLP.checkMaturity(uint256) (contracts/pools/LodaSLP.sol#71-78)
        - LodaSLP.claimAllRewards() (contracts/pools/LodaSLP.sol#382-401)
        - LodaSLP.claimRewards(uint256) (contracts/pools/LodaSLP.sol#366-382)
        - LodaSLP.computeRewardsForStake(uint256) (contracts/pools/LodaSLP.sol#351-362)
        - LodaSLP.getInvestorStakes() (contracts/pools/LodaSLP.sol#173-185)
        - LodaSLP.poolStakes (contracts/pools/LodaSLP.sol#45)
        - LodaSLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaSLP.sol#234-252)
        - LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229)
        - LodaSLP.updateStakeTime_FOR_TEST_REMOVE(uint256,uint256) (contracts/pools/LodaSLP.sol#483-495)
        - LodaSLP.withdraw(uint256) (contracts/pools/LodaSLP.sol#406-434)
        - replenishFunds(getTotalUsdc()) (contracts/pools/LodaSLP.sol#432-433)
                - unavailableUSDC = toBeUnavailable (contracts/pools/LodaSLP.sol#263-264)
        LodaSLP.unavailableUSDC (contracts/pools/LodaSLP.sol#38) can be used in cross function reentrancies:
        - LodaSLP.getTotalUsdc() (contracts/pools/LodaSLP.sol#171-173)
        - LodaSLP.poolSize() (contracts/pools/LodaSLP.sol#163-167)
        - LodaSLP.replenishFunds(uint256) (contracts/pools/LodaSLP.sol#258-264)
        - LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229)
        - LodaSLP.updateLiquidity(uint256) (contracts/pools/LodaSLP.sol#280-285)
        - LodaSLP.withdraw(uint256) (contracts/pools/LodaSLP.sol#406-434)
```



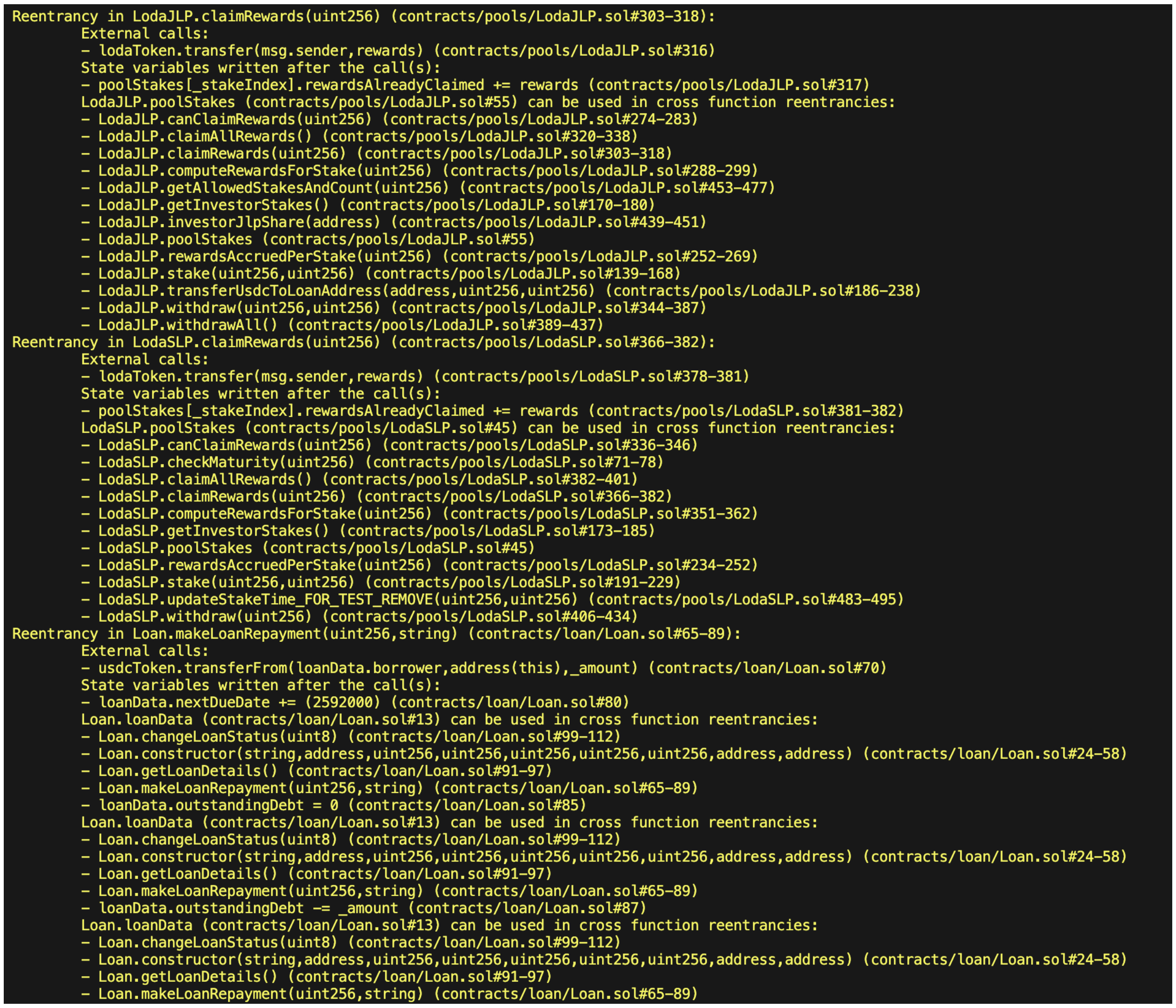
```
Reentrancy in LodaJLP.withdraw(uint256,uint256) (contracts/pools/LodaJLP.sol#344-387):
        External calls:
        - usdcToken.transfer(msg.sender,_amount) (contracts/pools/LodaJLP.sol#355)
        State variables written after the call(s):
        - poolStakes[_stakeId].amountAlreadyWithdrawn = alreadyWithdrawn + _amount (contracts/pools/LodaJLP.sol#356-358)
        LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55) can be used in cross function reentrancies:
        - LodaJLP.canClaimRewards(uint256) (contracts/pools/LodaJLP.sol#274-283)
        - LodaJLP.claimAllRewards() (contracts/pools/LodaJLP.sol#320-338)
        - LodaJLP.claimRewards(uint256) (contracts/pools/LodaJLP.sol#303-318)
        - LodaJLP.computeRewardsForStake(uint256) (contracts/pools/LodaJLP.sol#288-299)
        - LodaJLP.getAllowedStakesAndCount(uint256) (contracts/pools/LodaJLP.sol#453-477)
        - LodaJLP.getInvestorStakes() (contracts/pools/LodaJLP.sol#170-180)
        - LodaJLP.investorJlpShare(address) (contracts/pools/LodaJLP.sol#439-451)
        - LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55)
        - LodaJLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaJLP.sol#252-269)
        - LodaJLP.stake(uint256,uint256) (contracts/pools/LodaJLP.sol#139-168)
        - LodaJLP.transferUsdcToLoanAddress(address,uint256,uint256) (contracts/pools/LodaJLP.sol#186-238)
        - LodaJLP.withdraw(uint256,uint256) (contracts/pools/LodaJLP.sol#344-387)
        - LodaJLP.withdrawAll() (contracts/pools/LodaJLP.sol#389-437)
        - poolStakes[_stakeId].amountWithdrawable = amountWithdrawable - _amount (contracts/pools/LodaJLP.sol#359)
        LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55) can be used in cross function reentrancies:
        - LodaJLP.canClaimRewards(uint256) (contracts/pools/LodaJLP.sol#274-283)
        - LodaJLP.claimAllRewards() (contracts/pools/LodaJLP.sol#320-338)
        - LodaJLP.claimRewards(uint256) (contracts/pools/LodaJLP.sol#303-318)
        - LodaJLP.computeRewardsForStake(uint256) (contracts/pools/LodaJLP.sol#288-299)
        - LodaJLP.getAllowedStakesAndCount(uint256) (contracts/pools/LodaJLP.sol#453-477)
        - LodaJLP.getInvestorStakes() (contracts/pools/LodaJLP.sol#170-180)
        - LodaJLP.investorJlpShare(address) (contracts/pools/LodaJLP.sol#439-451)
        - LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55)
        - LodaJLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaJLP.sol#252-269)
        - LodaJLP.stake(uint256,uint256) (contracts/pools/LodaJLP.sol#139-168)
        - LodaJLP.transferUsdcToLoanAddress(address,uint256,uint256) (contracts/pools/LodaJLP.sol#186-238)
        - LodaJLP.withdraw(uint256,uint256) (contracts/pools/LodaJLP.sol#344-387)
        - LodaJLP.withdrawAll() (contracts/pools/LodaJLP.sol#389-437)
```



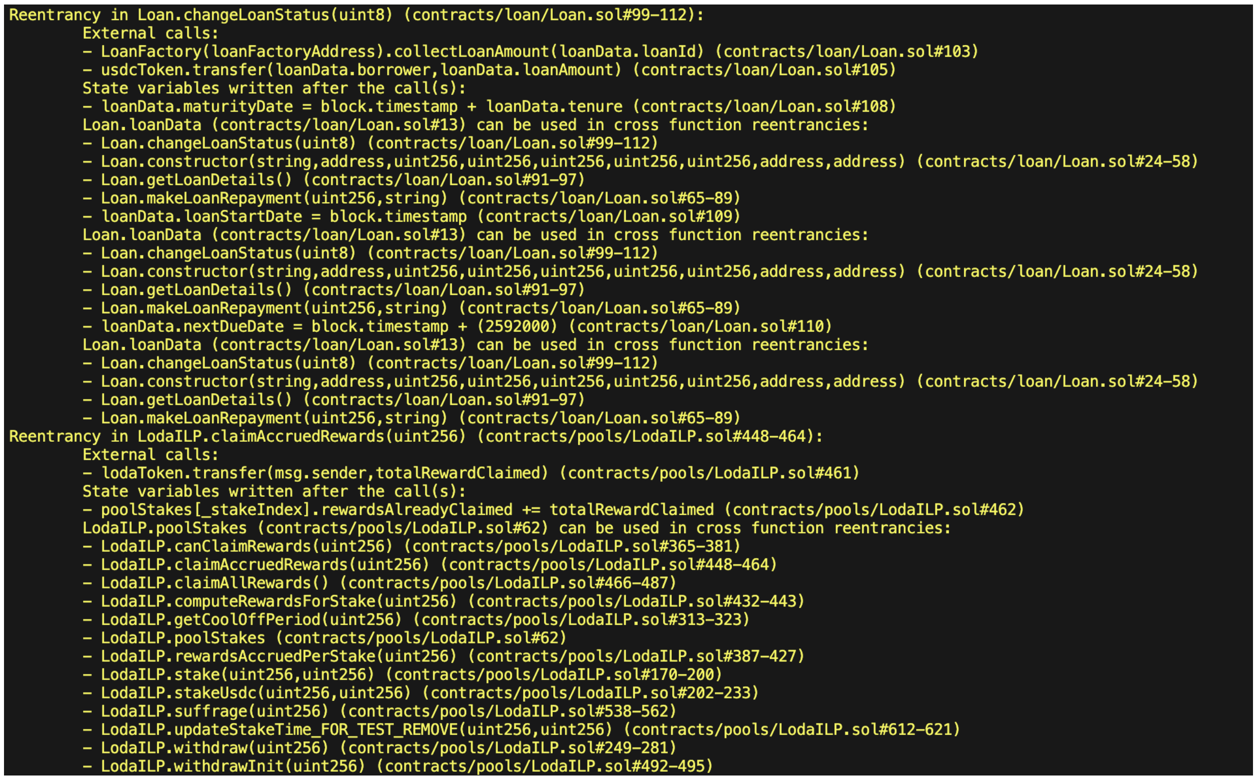
```
Reentrancy in LodaJLP.transferUsdcToLoanAddress(address,uint256,uint256) (contracts/pools/LodaJLP.sol#186-238):
        External calls:
        - usdcToken.transfer(_loanAddress,_amount) (contracts/pools/LodaJLP.sol#235)
        State variables written after the call(s):
        - availableUSDC -= _amount (contracts/pools/LodaJLP.sol#236)
        LodaJLP.availableUSDC (contracts/pools/LodaJLP.sol#21) can be used in cross function reentrancies:
        - LodaJLP.availableUSDC (contracts/pools/LodaJLP.sol#21)
        - LodaJLP.investorJlpShare(address) (contracts/pools/LodaJLP.sol#439-451)
        - LodaJLP.profitShare(uint256,uint256) (contracts/pools/LodaJLP.sol#240-247)
        - LodaJLP.stake(uint256,uint256) (contracts/pools/LodaJLP.sol#139-168)
        - LodaJLP.transferUsdcToLoanAddress(address,uint256,uint256) (contracts/pools/LodaJLP.sol#186-238)
Reentrancy in LodaILP.withdraw(uint256) (contracts/pools/LodaILP.sol#249-281):
        External calls:
        - lodestarToken.burn(msg.sender,_stake.atticusTokens) (contracts/pools/LodaILP.sol#272)
        - lodaToken.transfer(msg.sender,currentLodaAmount) (contracts/pools/LodaILP.sol#273)
        - lodaToken.transfer(msg.sender,claimableRewards) (contracts/pools/LodaILP.sol#276)
        State variables written after the call(s):
        - poolStakes[_stakeIndex].rewardsAlreadyClaimed += claimableRewards (contracts/pools/LodaILP.sol#277)
        LodaILP.poolStakes (contracts/pools/LodaILP.sol#62) can be used in cross function reentrancies:
        - LodaILP.canClaimRewards(uint256) (contracts/pools/LodaILP.sol#365-381)
        - LodaILP.claimAccruedRewards(uint256) (contracts/pools/LodaILP.sol#448-464)
        - LodaILP.claimAllRewards() (contracts/pools/LodaILP.sol#466-487)
        - LodaILP.computeRewardsForStake(uint256) (contracts/pools/LodaILP.sol#432-443)
        - LodaILP.getCoolOffPeriod(uint256) (contracts/pools/LodaILP.sol#313-323)
        - LodaILP.poolStakes (contracts/pools/LodaILP.sol#62)
        - LodaILP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaILP.sol#387-427)
        - LodaILP.stake(uint256,uint256) (contracts/pools/LodaILP.sol#170-200)
        - LodaILP.stakeUsdc(uint256,uint256) (contracts/pools/LodaILP.sol#202-233)
        - LodaILP.suffrage(uint256) (contracts/pools/LodaILP.sol#538-562)
        - LodaILP.updateStakeTime_FOR_TEST_REMOVE(uint256,uint256) (contracts/pools/LodaILP.sol#612-621)
        - LodaILP.withdraw(uint256) (contracts/pools/LodaILP.sol#249-281)
        - LodaILP.withdrawInit(uint256) (contracts/pools/LodaILP.sol#492-495)
        - poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN (contracts/pools/LodaILP.sol#280)
        LodaILP.poolStakes (contracts/pools/LodaILP.sol#62) can be used in cross function reentrancies:
        - LodaILP.canClaimRewards(uint256) (contracts/pools/LodaILP.sol#365-381)
        - LodaILP.claimAccruedRewards(uint256) (contracts/pools/LodaILP.sol#448-464)
        - LodaILP.claimAllRewards() (contracts/pools/LodaILP.sol#466-487)
        - LodaILP.computeRewardsForStake(uint256) (contracts/pools/LodaILP.sol#432-443)
        - LodaILP.getCoolOffPeriod(uint256) (contracts/pools/LodaILP.sol#313-323)
        - LodaILP.poolStakes (contracts/pools/LodaILP.sol#62)
        - LodaILP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaILP.sol#387-427)
        - LodaILP.stake(uint256,uint256) (contracts/pools/LodaILP.sol#170-200)
        - LodaILP.stakeUsdc(uint256,uint256) (contracts/pools/LodaILP.sol#202-233)
        - LodaILP.suffrage(uint256) (contracts/pools/LodaILP.sol#538-562)
        - LodaILP.updateStakeTime_FOR_TEST_REMOVE(uint256,uint256) (contracts/pools/LodaILP.sol#612-621)
        - LodaILP.withdraw(uint256) (contracts/pools/LodaILP.sol#249-281)
        - LodaILP.withdrawInit(uint256) (contracts/pools/LodaILP.sol#492-495)
```



```
Reentrancy in LodaILP.stake(uint256,uint256) (contracts/pools/LodaILP.sol#170-200):
        External calls:
        - lodaToken.transferFrom(msg.sender,address(this),_stake) (contracts/pools/LodaILP.sol#178)
        - IToken(atticusTokenAddress).mint(msg.sender,lodeStarAmount) (contracts/pools/LodaILP.sol#179)
        State variables written after the call(s):
        - totalCiceroStaked = totalCiceroStaked + _stake (contracts/pools/LodaILP.sol#180)
        LodaILP.totalCiceroStaked (contracts/pools/LodaILP.sol#30) can be used in cross function reentrancies:
        - LodaILP.getLodatoLodestar(uint256) (contracts/pools/LodaILP.sol#288-307)
        - LodaILP.getLodestarToLoda(uint256) (contracts/pools/LodaILP.sol#340-359)
        - LodaILP.stake(uint256,uint256) (contracts/pools/LodaILP.sol#170-200)
        - LodaILP.totalCiceroStaked (contracts/pools/LodaILP.sol#30)
Reentrancy in LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229):
        External calls:
        - usdcToken.transferFrom(msg.sender,address(this),_usdcStake) (contracts/pools/LodaSLP.sol#204-205)
        - IToken(slpTokenAddress).mint(msg.sender,slpAmount) (contracts/pools/LodaSLP.sol#205-206)
        State variables written after the call(s):
        - availableUSDC = toBeAvailable (contracts/pools/LodaSLP.sol#209-210)
        LodaSLP.availableUSDC (contracts/pools/LodaSLP.sol#37) can be used in cross function reentrancies:
        - LodaSLP.addUSDC_FOR_TEST_REMOVE(uint256) (contracts/pools/LodaSLP.sol#495-499)
        - LodaSLP.defaultPayments(uint256) (contracts/pools/LodaSLP.sol#452-462)
        - LodaSLP.getTotalUsdc() (contracts/pools/LodaSLP.sol#171-173)
        - LodaSLP.poolSize() (contracts/pools/LodaSLP.sol#163-167)
        - LodaSLP.profitShare(uint256,uint256) (contracts/pools/LodaSLP.sol#302-313)
        - LodaSLP.removeUSDC_FOR_TEST_REMOVE(uint256) (contracts/pools/LodaSLP.sol#499-503)
        - LodaSLP.replenishFunds(uint256) (contracts/pools/LodaSLP.sol#258-264)
        - LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229)
        - LodaSLP.transferUsdcToLoanAddress(address,uint256) (contracts/pools/LodaSLP.sol#434-452)
        - LodaSLP.updateLiquidity(uint256) (contracts/pools/LodaSLP.sol#280-285)
        - unavailableUSDC = toBeUnavailable (contracts/pools/LodaSLP.sol#210-211)
        LodaSLP.unavailableUSDC (contracts/pools/LodaSLP.sol#38) can be used in cross function reentrancies:
        - LodaSLP.getTotalUsdc() (contracts/pools/LodaSLP.sol#171-173)
        - LodaSLP.poolSize() (contracts/pools/LodaSLP.sol#163-167)
        - LodaSLP.replenishFunds(uint256) (contracts/pools/LodaSLP.sol#258-264)
        - LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229)
        - LodaSLP.updateLiquidity(uint256) (contracts/pools/LodaSLP.sol#280-285)
        - LodaSLP.withdraw(uint256) (contracts/pools/LodaSLP.sol#406-434)
Reentrancy in LodaILP.stakeUsdc(uint256,uint256) (contracts/pools/LodaILP.sol#202-233):
        External calls:
        - usdcToken.transferFrom(msg.sender,address(this),_usdcAmount) (contracts/pools/LodaILP.sol#210)
        - IToken(atticusTokenAddress).mint(msg.sender,lodestarTokens) (contracts/pools/LodaILP.sol#212)
        State variables written after the call(s):
        - usdcInPool += _usdcAmount (contracts/pools/LodaILP.sol#227)
        LodaILP.usdcInPool (contracts/pools/LodaILP.sol#31) can be used in cross function reentrancies:
        - LodaILP.getLodatoLodestar(uint256) (contracts/pools/LodaILP.sol#288-307)
        - LodaILP.getLodestarToLoda(uint256) (contracts/pools/LodaILP.sol#340-359)
        - LodaILP.profitShare(uint256,uint256) (contracts/pools/LodaILP.sol#574-581)
        - LodaILP.stakeUsdc(uint256,uint256) (contracts/pools/LodaILP.sol#202-233)
        - LodaILP.usdcInPool (contracts/pools/LodaILP.sol#31)
```



```
Reentrancy in LodaJLP.claimRewards(uint256) (contracts/pools/LodaJLP.sol#303-318):
	External calls:
	- lodaToken.transfer(msg.sender,rewards) (contracts/pools/LodaJLP.sol#316)
	State variables written after the call(s):
	- poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards (contracts/pools/LodaJLP.sol#317)
	LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55) can be used in cross function reentrancies:
	- LodaJLP.canClaimRewards(uint256) (contracts/pools/LodaJLP.sol#274-283)
	- LodaJLP.claimAllRewards() (contracts/pools/LodaJLP.sol#320-338)
	- LodaJLP.claimRewards(uint256) (contracts/pools/LodaJLP.sol#303-318)
	- LodaJLP.computeRewardsForStake(uint256) (contracts/pools/LodaJLP.sol#288-299)
	- LodaJLP.getAllowedStakesAndCount(uint256) (contracts/pools/LodaJLP.sol#453-477)
	- LodaJLP.getInvestorStakes() (contracts/pools/LodaJLP.sol#170-180)
	- LodaJLP.investorJlpShare(address) (contracts/pools/LodaJLP.sol#439-451)
	- LodaJLP.poolStakes (contracts/pools/LodaJLP.sol#55)
	- LodaJLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaJLP.sol#252-269)
	- LodaJLP.stake(uint256,uint256) (contracts/pools/LodaJLP.sol#139-168)
	- LodaJLP.transferUsdcToLoanAddress(address,uint256,uint256) (contracts/pools/LodaJLP.sol#186-238)
	- LodaJLP.withdraw(uint256,uint256) (contracts/pools/LodaJLP.sol#344-387)
	- LodaJLP.withdrawAll() (contracts/pools/LodaJLP.sol#389-437)
Reentrancy in LodaSLP.claimRewards(uint256) (contracts/pools/LodaSLP.sol#366-382):
	External calls:
	- lodaToken.transfer(msg.sender,rewards) (contracts/pools/LodaSLP.sol#378-381)
	State variables written after the call(s):
	- poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards (contracts/pools/LodaSLP.sol#381-382)
	LodaSLP.poolStakes (contracts/pools/LodaSLP.sol#45) can be used in cross function reentrancies:
	- LodaSLP.canClaimRewards(uint256) (contracts/pools/LodaSLP.sol#336-346)
	- LodaSLP.checkMaturity(uint256) (contracts/pools/LodaSLP.sol#71-78)
	- LodaSLP.claimAllRewards() (contracts/pools/LodaSLP.sol#382-401)
	- LodaSLP.claimRewards(uint256) (contracts/pools/LodaSLP.sol#366-382)
	- LodaSLP.computeRewardsForStake(uint256) (contracts/pools/LodaSLP.sol#351-362)
	- LodaSLP.getInvestorStakes() (contracts/pools/LodaSLP.sol#173-185)
	- LodaSLP.poolStakes (contracts/pools/LodaSLP.sol#45)
	- LodaSLP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaSLP.sol#234-252)
	- LodaSLP.stake(uint256,uint256) (contracts/pools/LodaSLP.sol#191-229)
	- LodaSLP.updateStakeTime_FOR_TEST_REMOVE(uint256,uint256) (contracts/pools/LodaSLP.sol#483-495)
	- LodaSLP.withdraw(uint256) (contracts/pools/LodaSLP.sol#406-434)
Reentrancy in Loan.makeLoanRepayment(uint256,string) (contracts/loan/Loan.sol#65-89):
	External calls:
	- usdcToken.transferFrom(loanData.borrower,address(this),_amount) (contracts/loan/Loan.sol#70)
	State variables written after the call(s):
	- loanData.nextDueDate += (2592000) (contracts/loan/Loan.sol#80)
	Loan.loanData (contracts/loan/Loan.sol#13) can be used in cross function reentrancies:
	- Loan.changeLoanStatus(uint8) (contracts/loan/Loan.sol#99-112)
	- Loan.constructor(string,address,uint256,uint256,uint256,uint256,uint256,address,address) (contracts/loan/Loan.sol#24-58)
	- Loan.getLoanDetails() (contracts/loan/Loan.sol#91-97)
	- Loan.makeLoanRepayment(uint256,string) (contracts/loan/Loan.sol#65-89)
	- loanData.outstandingDebt = 0 (contracts/loan/Loan.sol#85)
	Loan.loanData (contracts/loan/Loan.sol#13) can be used in cross function reentrancies:
	- Loan.changeLoanStatus(uint8) (contracts/loan/Loan.sol#99-112)
	- Loan.constructor(string,address,uint256,uint256,uint256,uint256,uint256,address,address) (contracts/loan/Loan.sol#24-58)
	- Loan.getLoanDetails() (contracts/loan/Loan.sol#91-97)
	- Loan.makeLoanRepayment(uint256,string) (contracts/loan/Loan.sol#65-89)
	- loanData.outstandingDebt -= _amount (contracts/loan/Loan.sol#87)
	Loan.loanData (contracts/loan/Loan.sol#13) can be used in cross function reentrancies:
	- Loan.changeLoanStatus(uint8) (contracts/loan/Loan.sol#99-112)
	- Loan.constructor(string,address,uint256,uint256,uint256,uint256,uint256,address,address) (contracts/loan/Loan.sol#24-58)
	- Loan.getLoanDetails() (contracts/loan/Loan.sol#91-97)
	- Loan.makeLoanRepayment(uint256,string) (contracts/loan/Loan.sol#65-89)
```



```
Reentrancy in Loan.changeLoanStatus(uint8) (contracts/loan/Loan.sol#99-112):
        External calls:
        - LoanFactory(loanFactoryAddress).collectLoanAmount(loanData.loanId) (contracts/loan/Loan.sol#103)
        - usdcToken.transfer(loanData.borrower,loanData.loanAmount) (contracts/loan/Loan.sol#105)
        State variables written after the call(s):
        - loanData.maturityDate = block.timestamp + loanData.tenure (contracts/loan/Loan.sol#108)
        Loan.loanData (contracts/loan/Loan.sol#13) can be used in cross function reentrancies:
        - Loan.changeLoanStatus(uint8) (contracts/loan/Loan.sol#99-112)
        - Loan.constructor(string,address,uint256,uint256,uint256,uint256,uint256,address,address) (contracts/loan/Loan.sol#24-58)
        - Loan.getLoanDetails() (contracts/loan/Loan.sol#91-97)
        - Loan.makeLoanRepayment(uint256,string) (contracts/loan/Loan.sol#65-89)
        - loanData.loanStartDate = block.timestamp (contracts/loan/Loan.sol#109)
        Loan.loanData (contracts/loan/Loan.sol#13) can be used in cross function reentrancies:
        - Loan.changeLoanStatus(uint8) (contracts/loan/Loan.sol#99-112)
        - Loan.constructor(string,address,uint256,uint256,uint256,uint256,uint256,address,address) (contracts/loan/Loan.sol#24-58)
        - Loan.getLoanDetails() (contracts/loan/Loan.sol#91-97)
        - Loan.makeLoanRepayment(uint256,string) (contracts/loan/Loan.sol#65-89)
        - loanData.nextDueDate = block.timestamp + (2592000) (contracts/loan/Loan.sol#110)
        Loan.loanData (contracts/loan/Loan.sol#13) can be used in cross function reentrancies:
        - Loan.changeLoanStatus(uint8) (contracts/loan/Loan.sol#99-112)
        - Loan.constructor(string,address,uint256,uint256,uint256,uint256,uint256,address,address) (contracts/loan/Loan.sol#24-58)
        - Loan.getLoanDetails() (contracts/loan/Loan.sol#91-97)
        - Loan.makeLoanRepayment(uint256,string) (contracts/loan/Loan.sol#65-89)
Reentrancy in LodaILP.claimAccruedRewards(uint256) (contracts/pools/LodaILP.sol#448-464):
        External calls:
        - lodaToken.transfer(msg.sender,totalRewardClaimed) (contracts/pools/LodaILP.sol#461)
        State variables written after the call(s):
        - poolStakes[_stakeIndex].rewardsAlreadyClaimed += totalRewardClaimed (contracts/pools/LodaILP.sol#462)
        LodaILP.poolStakes (contracts/pools/LodaILP.sol#62) can be used in cross function reentrancies:
        - LodaILP.canClaimRewards(uint256) (contracts/pools/LodaILP.sol#365-381)
        - LodaILP.claimAccruedRewards(uint256) (contracts/pools/LodaILP.sol#448-464)
        - LodaILP.claimAllRewards() (contracts/pools/LodaILP.sol#466-487)
        - LodaILP.computeRewardsForStake(uint256) (contracts/pools/LodaILP.sol#432-443)
        - LodaILP.getCoolOffPeriod(uint256) (contracts/pools/LodaILP.sol#313-323)
        - LodaILP.poolStakes (contracts/pools/LodaILP.sol#62)
        - LodaILP.rewardsAccruedPerStake(uint256) (contracts/pools/LodaILP.sol#387-427)
        - LodaILP.stake(uint256,uint256) (contracts/pools/LodaILP.sol#170-200)
        - LodaILP.stakeUsdc(uint256,uint256) (contracts/pools/LodaILP.sol#202-233)
        - LodaILP.suffrage(uint256) (contracts/pools/LodaILP.sol#538-562)
        - LodaILP.updateStakeTime_FOR_TEST_REMOVE(uint256,uint256) (contracts/pools/LodaILP.sol#612-621)
        - LodaILP.withdraw(uint256) (contracts/pools/LodaILP.sol#249-281)
        - LodaILP.withdrawInit(uint256) (contracts/pools/LodaILP.sol#492-495)
```

4.4 Unrestricted Loan Creation

Severity	Status	Type of Analysis
● Critical	Identified	Dynamic

Contract Name:

loanFactory.sol

Description:

The createLoanContract function allows any user to create a loan by calling the function and setting parameters such as interest rate, tenure, and monthly EMI. The function does not have any restrictions on who can call it.

Locations:

function createLoanContract: Line no = 69 - 101

Remediation:

- Restrict the ability to create loans to only authorised users, such as the contract owner or approved borrowers.

Impact:

Allowing unrestricted loan creation poses significant risks:

As per the protocol, the loan is only given on institutional level here but here neither we have any parameter to check for that nor the owner or any administrator is checking.

- Malicious users can set unreasonable interest rates or tenure periods.
- Attackers can flood the system with numerous loans, overwhelming the platform and causing potential service disruptions.

Code Snippet:

```
            function createLoanContract(

        string memory instituitionName,
        address borrower,

        uint256 loanAmount,

        uint256 tenure,

        uint256 interestRate,

        uint256 monthlyEmi

    ) public returns (uint256, address) {

        Loan loan = new Loan(

            instituitionName,

            borrower,

            loanAmount,

            tenure,

            loanCount.current(),

            interestRate,

            monthlyEmi,

            usdcTokenAddress,

            address(this)

        );



        address loanAddress = address(loan);

        loanIdToAddressMapping[loanCount.current()] = loanAddress;



        borrowerAddressToLoanIdMapping[borrower][

            userLoanCount[borrower]

        ] = loanCount.current();

        loanDetails[loanCount.current()] = loan;

        ++userLoanCount[borrower];

        emit LoanCreated(borrower, loanAddress);

        uint256 loanId = loanCount.current();

        loanCount.increment();

        return (loanId, loanAddress);
        return (loanId, loanAddress);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.5 Centralization Risk for trusted owners

Severity	Status	Type of Analysis
● High	Identified	Static

Contract Name:

LoanFactory.sol, ILPCompute.sol, LodaILP.sol, MockedLodaSLP.sol, USDCToken.sol, Atticus.sol, CiceroNFT.sol, CiceroToken.sol, SLPToken.sol,

Description:

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Locations:

- **LoanFactory.sol:** 41, 48, 55, 64
- **ILPCompute.sol:** 27
- **LodaILP.sol:** 102, 113, 123, 133, 144, 154, 162
- **LodaJLP.sol:** 79, 86, 110, 120, 130
- **LodaSLP.sol:** 57, 107, 117, 122, 131, 141, 151, 161
- **MockedLodaSLP.sol:** 7, 13, 20
- **USDCToken.sol:** 30, 34, 38, 42
- **Atticus.sol:** 32, 36, 40, 44
- **CiceroNFT.sol:** 40, 49, 53, 77
- **CiceroToken.sol:** 32, 36
- **SLPToken.sol:** 30, 34, 38, 42

Remediation:

To mitigate centralization risks in smart contracts, implement multisignature (multisig) wallets requiring multiple approvals for critical actions and introduce timelock mechanisms that delay administrative changes, allowing community review. Additionally, adopt decentralized governance models to distribute decision-making among token holders or a DAO, and conduct regular third-party code audits with transparent public reporting to enhance security, trust, and resilience in the contract ecosystem.

Impact:

Centralization risk in smart contracts arises when a single or a few trusted owners possess privileged rights, such as the ability to perform administrative tasks or update contract parameters.Trusted owners could potentially exploit their privileges to perform malicious updates, drain funds, or alter contract functionality in ways that benefit themselves at the expense of other users.

Code Snippet:

LoanFactory.sol:

```
function updateLodaJLPAddress(address _lodaJLPAddress) external onlyOwner {

lodaJLPAddress = _lodaJLPAddress;

}



function updateLodaSLPAddress(address _lodaSLPAddress) external onlyOwner {

lodaSLPAddress = _lodaSLPAddress;

}



function updateLodaILPAddress(address _lodaILPAddress) external onlyOwner {

lodaILPAddress = _lodaILPAddress;

}



function updateUSDCTokenAddress(address _usdcTokenAddress)

external

onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;

}
```

ILPCompute.sol:

```
function updateRealMathAddress(address _realMathAddress)

external

onlyOwner

{

realMathAddress = _realMathAddress;

}
```

LoadILP.sol:

```
function changeRewardRate(uint256 maturityPeriodInDays, uint256 newRate)

public

onlyOwner

{

rewardRate[maturityPeriodInDays] = newRate;

emit RewardRateChanged(maturityPeriodInDays, newRate);

}


function updateUSDCTokenAddress(address _usdcTokenAddress)

external

onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;

}


function updateAtticusToken(address _atticusTokenAddress)

external

onlyOwner

{

atticusTokenAddress = _atticusTokenAddress;

}


function updateILPComputeAddress(address _ILPComputeAddress)

external

onlyOwner

{

ILPComputeAddress = _ILPComputeAddress;

}
```

```
function updateCiceroToken(address _ciceroTokenAddress)

external

checkZeroAddress(_ciceroTokenAddress)

onlyOwner

{

ciceroTokenAddress = _ciceroTokenAddress;

}


function updateSuffrageDecimation(uint256 _newDecimation)

external

onlyOwner

{

suffrageDecimation = _newDecimation;

}


function updateSLPPoolAddress(address _SLPPoolAddress) external onlyOwner {

SLPPoolAddress = _SLPPoolAddress;

}


function updateSLPPoolAddress(address _SLPPoolAddress) external onlyOwner {

SLPPoolAddress = _SLPPoolAddress;

}
```

LodaJLP.sol:

```
function updateCiceroTokenAddress(address _ciceroTokenAddress)

external

onlyOwner

{
```

```
ciceroTokenAddress = _ciceroTokenAddress;

}



function changeRewardRate(uint256 maturityPeriodInDays, uint256 newRate)

public

onlyOwner

{

rewardRate[maturityPeriodInDays] = newRate;

emit RewardRateChanged(maturityPeriodInDays, newRate);

}



function updatenftTokenAddress(address _nftTokenAddress)

external

onlyOwner

{

nftTokenAddress = _nftTokenAddress;

}



function updateLoanFactoryAddress(address _loanFactoryAddress)

external

onlyOwner

{

loanFactoryAddress = _loanFactoryAddress;

}



function updateUSDCTokenAddress(address _usdcTokenAddress)

external

onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;
```

```
}
```

LoadSLP.sol:

```
function changeRewardRate(uint256 maturityPeriodInMonths, uint256 newRate)

public

onlyOwner

{

rewardRate[maturityPeriodInMonths] = newRate;

emit RewardRateChanged(maturityPeriodInMonths, newRate);

}



function setPerSecondSLPRewards(uint256 _perSecondSLPRewardInUnitLoda)

external

onlyOwner

{

perSecondSLPRewardInUnitLoda = _perSecondSLPRewardInUnitLoda;

}



function updateUSDCTokenAddress(address _usdcTokenAddress)

external

onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;

}



function updateInitialStakeRatio(uint256 _ratio) external onlyOwner {

initialStakeRatio = _ratio;

}



function updateSLPTokenAddress(address _slpTokenAddress)
function updateSLPTokenAddress(address _slpTokenAddress)

external

onlyOwner

{

slpTokenAddress = _slpTokenAddress;

}



function updateCiceroTokenAddress(address _ciceroTokenAddress)

external

onlyOwner

{

ciceroTokenAddress = _ciceroTokenAddress;

}



function updateSLPComputeAddress(address _slpComputeAddress)

external

onlyOwner

{

slpComputeAddress = _slpComputeAddress;

}



function updateLoanFactoryAddress(address _loanFactoryAddress)

external

onlyOwner

{

loanFactoryAddress = _loanFactoryAddress;

}
```

MockedLodaSLP:

```
function updateAvailableUsdc(uint256 _availableUsdc) external onlyOwner {

availableUSDC = _availableUsdc;

}



function updateUnAvailableUsdc(uint256 _unAvailableUsdc)

external

onlyOwner

{

unavailableUSDC = _unAvailableUsdc;

}



function updateStakeStatus(uint256 _stakeIndex, uint256 _status)

external

onlyOwner

{

poolStakes[_stakeIndex].status = StakeCommons.StakeStatus(_status);

}
```

USDCToken:

```
contract USDCToken is

Initializable,

ERC20Upgradeable,

ERC20BurnableUpgradeable,

PausableUpgradeable,

OwnableUpgradeable,

ERC20PermitUpgradeable

{

/// @custom:oz-upgrades-unsafe-allow constructor

constructor() initializer {}

function initialize() public initializer {
```

```
__ERC20_init("USDCToken", "USDC");

__ERC20Burnable_init();

__Pausable_init();

__Ownable_init();

__ERC20Permit_init("USDC");

}

function pause() public onlyOwner {

_pause();

}

function unpause() public onlyOwner {

_unpause();

}

function mint(address to, uint256 amount) public whenNotPaused onlyOwner {

_mint(to, amount);

}

function burn(address from, uint256 amount) public whenNotPaused onlyOwner {

_burn(from, amount);

}

function _mint(address to, uint256 amount)

internal

override(ERC20Upgradeable)

{

super._mint(to, amount);

}

function _burn(address account, uint256 amount)

internal

override(ERC20Upgradeable)

{

super._burn(account, amount);
```

```
}

function decimals() public pure override returns (uint8) {

return 6;

}

}
```

Atticus.sol:

```
function pause() public onlyOwner {

_pause();

}


function unpause() public onlyOwner {

_unpause();

}


function mint(address to, uint256 amount) public onlyOwner {

_mint(to, amount);

}


function burn(address from, uint256 amount) public onlyOwner {

_burn(from, amount);

}
```

CiceroNFT.sol:

```
function updateLodaJLPAddress(address _lodaJLPAddress) external onlyOwner {

lodaJLPAddress = _lodaJLPAddress;

}


modifier checkCallerAddress() {

require(msg.sender == lodaJLPAddress, "unauthorized call");
_;

}



function pause() public onlyOwner {

_pause();

}



function unpause() public onlyOwner {

_unpause();

}



function safeMint(address to, uint256 tokenId) public checkCallerAddress {

_safeMint(to, tokenId);

}
```

<u>CiceroToken.sol:</u>

```
function pause() public onlyOwner {

_pause();

}



function unpause() public onlyOwner {

_unpause();

}
```

<u>SLPToken.sol:</u>

```
function pause() public onlyOwner {

_pause();

}



function unpause() public onlyOwner {
```

```
function unpause() public onlyOwner {

_unpause();

}



function mint(address to, uint256 amount) public whenNotPaused onlyOwner {

_mint(to, amount);

}



function burn(address from, uint256 amount) public whenNotPaused onlyOwner {

_burn(from, amount);

}
```

Reference:

https://rekt.news/munchables-rekt/

Proof of Vulnerability:

N.A.

4.6 Fixed JLP Share Allocation

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

loanFactory.sol

Description:

The collectLoanAmount function currently checks whether it is being called by the loan address and then allocates a fixed 20% of the loan amount from the Junior Liquidity Pool (JLP) without considering the actual available funds in JLP. This can lead to inefficiencies in fund allocation between JLP and the Senior Liquidity Pool (SLP).

Locations:

function collectLoanAmount: Line no = 131-143

Remediation:

- Dynamic Allocation: Adjust the function to dynamically calculate the amount to be taken from the JLP and the SLP based on available funds.

Impact:

- Inefficient Fund Utilization: If the JLP does not have sufficient funds to cover the fixed 20%, the function does not dynamically adjust the amount to be taken from the JLP and the SLP. This can lead to inefficiencies and potential fund shortages.

Code Snippet:

```
        function collectLoanAmount(uint256 _loanId) public {

        address loanAddress = loanIdToAddressMapping[_loanId];

        require(msg.sender == loanAddress, "Unauthorized call");

        Loan targetLoan = loanDetails[_loanId];

        LoanCommon.LoanDetails memory loan = targetLoan.getLoanDetails();

        uint256 amount = loan.loanAmount;

        uint256 maturityDate = loan.maturityDate;

        // calculate 20% of total loan amount

        uint256 jlpShare = amount / 5;

        getFirstLossCapital(loanAddress, jlpShare, maturityDate);

        getSlpFunds(loanAddress, amount - jlpShare);

        Loan(loanAddress).changeLoanStatus(2);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.7 Incomplete and Unregulated Loan Repayment Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

loanFactory.sol

Description:

The makeLoanRepayment function allows borrowers to repay their loans. It takes the loan ID, repayment amount, fine amount etc. However, it lacks essential checks and regulations, such as verifying if the repayment amount is equal to or greater than the monthly payment, implementing clear rules for fine calculation, and transferring fines to the treasury.

Locations:

function makeLoanRepayment: Line no = 145-162

Remediation:

- **Payment Amount Verification:** Add a check to ensure the repayment amount is equal to or greater than the monthly EMI
- **Fine Regulation:** Implement rules for calculating fines and validate the fine amount during repayment.

Impact:

- Incorrect Payments: Borrowers might underpay, causing discrepancies in loan balances.
- Inconsistent Fine Handling: Without clear fine regulations, fines may be applied arbitrarily or incorrectly.
- Incomplete Implementation: The lack of fine transfer to the treasury leaves the function incomplete.

Code Snippet:

```
        function makeLoanRepayment(

        uint256 _loanId,

        uint256 _amount,

        uint256 _fine,

        string memory _selection

    ) public {

        address loanAddress = loanIdToAddressMapping[_loanId];

        Loan(loanAddress).makeLoanRepayment(_amount, _selection);

        fines[_loanId] = _fine;

        Loan targetLoan = loanDetails[_loanId];

        LoanCommon.LoanDetails memory loan = targetLoan.getLoanDetails();

        uint256 remainingBalance = loan.outstandingDebt;

        if (remainingBalance == 0 || loan.outstandingDebt < loan.monthlyEmi) {

            Loan(loanAddress).changeLoanStatus(3);

            allocateProfitsToPools(_loanId);

        }

        //TODO transfer _fine to treasury contract

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.8 Lack of Daily Withdrawal Limit Check in withdraw Function

Severity	Status	Type of Analysis
● High	Identified	Static

Contract Name:

LodaSLP.sol

Description:

The withdraw function facilitates the withdrawal of USDC that a user has staked, along with burning corresponding SLP tokens. It also triggers reward claims if available. However, it lacks a crucial check to enforce a daily withdrawal limit for users.

Locations:

function withdraw: Line no= 407-435

Remediation:

- **Implement Daily Withdrawal Limit:** Introduce a mechanism to enforce a daily withdrawal limit based on protocol liquidity or other predefined criteria.
- **Enhance Liquidity Management:** Ensure withdrawals do not exceed the available liquidity reserve, considering both available and unavailable USDC balances.

Impact:

Impact:

- The absence of a daily withdrawal limit check exposes the protocol to potential liquidity risks and operational inefficiencies. Without this check, users could withdraw more USDC than the protocol can sustainably support within a given timeframe.
- Protocol is simply making the availableUSDC(Used for loan) convert into unavailableUSDC(It is liquid funds, these are used when investor wants to withdraw stake).

Attack:

- **Excessive Withdrawals:** Users might withdraw more USDC than sustainable by the protocol’s liquidity reserve, causing temporary or prolonged liquidity shortages.
- **Protocol Stability:** Imbalance between available and unavailable USDC due to unchecked withdrawals could undermine the protocol's ability to fulfill loan obligations or maintain liquidity.

- **Single User:** Single user can withdraw exccessively and then funds won't be available for other investors.

Code Snippet:

```
        function withdraw(uint256 _stakeIndex) public checkMaturity(_stakeIndex) {

        Stake memory _stake = poolStakes[_stakeIndex];

        uint256 slpAmount = _stake.slpTokens;

        uint256 usdcAmount = slpToUsdc(slpAmount);

        require(

            _stake.status != StakeCommons.StakeStatus.WITHDRAWN,

            "Stake has already been withdrawn"

        );

        require(usdcAmount > 0, "Should be a non zero transfer");

        require(

            hasEnoughLiquidity(usdcAmount),

            "Not enough USDC to withdraw in the pool"

        );

        updateLiquidity(usdcAmount);

        unavailableUSDC -= usdcAmount;

        IToken usdcToken = IToken(usdcTokenAddress);

        IToken slpToken = IToken(slpTokenAddress);

        usdcToken.transfer(msg.sender, usdcAmount);

        slpToken.burn(msg.sender, slpAmount);

        if (

            IToken(ciceroTokenAddress).balanceOf(address(this)) >=

            computeRewardsForStake(_stakeIndex)

        ) {

            claimRewards(_stakeIndex);

        }

        poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN;

        replenishFunds(getTotalUsdc());

        emit USDCWithdrawn(msg.sender, usdcAmount);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.9 Missing Check for EMI Amount in makeLoanRepayment Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

loan.sol

Description:

The makeLoanRepayment function does not verify whether the repayment amount is equal to the monthly EMI. This check is essential to ensure that the borrower is making the correct payment.

Locations:

function makeLoanRepayment: Line no = 65-89

Remediation:

- Add a check to ensure the amount being paid is equal to the monthly EMI.

Impact:

- Borrowers might make partial payments or incorrect amounts, leading to inconsistencies in loan repayments.

Code Snippet:

```
        function makeLoanRepayment(uint256 _amount, string memory _selection)
        public
        checkLoanFactoryAddress
    {
        IToken usdcToken = IToken(usdcTokenAddress);
        usdcToken.transferFrom(loanData.borrower, address(this), _amount);
        repaymentCounter.increment();

        countToRepayment[repaymentCounter.current()] = LoanCommon.Repayment({
            dueDate: loanData.nextDueDate,
            paymentDate: block.timestamp,
            amount: _amount,
            selection: _selection
        });

        loanData.nextDueDate += (30 days);
        if (
            keccak256(abi.encodePacked(_selection)) ==
            keccak256(abi.encodePacked("totalPayment"))
        ) {
            loanData.outstandingDebt = 0;
        } else {
            loanData.outstandingDebt -= _amount;
        }
    }
```

Reference:

Proof of Vulnerability:

N.A.

4.10 Ownership access not being checked in profitShare Function

Severity	Status	Type of Analysis
🔴 High	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The profitShare function is intended to record profits earned from a specific loan and update the available USDC balance, It is called by LoanFactorry.sol contract when the profits needs to be distributed. However, it lacks access control mechanisms, allowing any user to call it.

Locations:

function profitShare: Line no = 574-581

Remediation:

- Access Control: Restrict access to the profitShare function using access modifiers (private or internal) or implement role-based access controls to ensure only authorized contracts or roles can update profit information.

Impact:

Impact:

As this is public function so it can be called by anyone, so this vulnerability introduces several risks to the protocol's integrity:

- **Unauthorized Access:** Without access controls, any user can manipulate the earnedProfit mapping for arbitrary loanId.

Attack:

- **Fake Profits Injection:** Malicious users can fabricate profits if they know loanId by invoking the function with arbitrary _loanId and _amount values, undermining the accuracy of profit reporting, thus disturbing the liquidity pool.

Code Snippet:

```
        function profitShare(uint256 _loanId, uint256 _amount) public {

        earnedProfit[_loanId] = LoanCommon.Profit({

            amount: _amount,

            date: block.timestamp

        });


        usdcInPool += _amount;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.11 Ownership access not being checked in profitShare Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The profitShare function is intended to record profits earned from a specific loan and update the available USDC balance, It is called by LoanFactorry.sol contract when the profits needs to be distributed. However, it lacks access control mechanisms, allowing any user to call it.

Locations:

function profitShare: Line no = 240 - 247

Remediation:

- **Access Control:** Restrict access to the profitShare function using access modifiers (private or internal) or implement role-based access controls to ensure only authorized contracts or roles can update profit information.

Impact:

Impact:

As this is public function so it can be called by anyone, so this vulnerability introduces several risks to the protocol's integrity and transparency:

- **Unauthorized Access:** Without access controls, any user can manipulate the earnedProfit mapping for any loanId.

Attack:

- **Fake Profits Injection:** Malicious users can fabricate profits by invoking the function with arbitrary _loanId and _amount values, undermining the accuracy of profit reporting, thus disturbing the liquidity pool.

Code Snippet:

```
        function profitShare(uint256 _loanId, uint256 _amount) public {

        EarnedProfit[_loanId] = LoanCommon.Profit({

            amount: _amount,

            date: block.timestamp

        });



        availableUSDC += _amount;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.12 Ownership access not being checked in profitShare Function

Severity	Status	Type of Analysis
🔴 High	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The profitShare function is intended to record profits earned from a specific loan and update the available USDC balance, It is called by LoanFactorry.sol contract when the profits needs to be distributed. However, it lacks access control mechanisms, allowing any user to call it.

Locations:

funciton profitShare: Line no = 307 - 314

Remediation:

- Access Control: Restrict access to the profitShare function using access modifiers (private or internal) or implement role-based access controls to ensure only authorized contracts or roles can update profit information.

Impact:

Impact:

As this is public function so it can be called by anyone, so this vulnerability introduces several risks to the protocol's integrity and transparency:

- **Unauthorized Access:** Without access controls, any user can manipulate the earnedProfit mapping for any loanId.

Attack:

- **Fake Profits Injection:** Malicious users if they know the loanid can fabricate profits by invoking the function with arbitrary _loanId and _amount values, undermining the accuracy of profit reporting, thus disturbing the liquidity pool.

Code Snippet:

```
        function profitShare(uint256 _loanId, uint256 _amount) public {

        earnedProfit[_loanId] = LoanCommon.Profit({

            amount: _amount,

            date: block.timestamp

        });



        availableUSDC += _amount;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.13 Ownership access not being checked of the stake index in Claim Rewards Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The claimRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.

Locations:

function claimRewards: Line no = 303 - 318

Remediation:

- Ownership Verification: Implement checks to ensure that msg.sender is the owner of the stake identified by _stakeIndex before allowing reward claims.

Impact:

Impact:

- This function's lack of ownership verification so without verifying ownership , potentially leading to unauthorized reward claims.

Attack:

- **Unauthorized Claim:** Any caller can execute claimRewards if they know _stakeIndex, irrespective of stake ownership, leading to unauthorized withdrawal of rewards.
- **Stake Status Integrity:** Failure to update lastClaimedAt may misrepresent the last claimed time which isn't correct.

Code Snippet:

```
        function claimRewards(uint256 _stakeIndex) public {

        require(

            canClaimRewards(_stakeIndex),

            "Can't withdraw this stake withdraw"

        );

        IToken lodaToken = IToken(ciceroTokenAddress);

        uint256 rewards = computeRewardsForStake(_stakeIndex);


        require(

            lodaToken.balanceOf(address(this)) >= rewards,

            "Insufficient LODA Token"

        );


        lodaToken.transfer(msg.sender, rewards);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.14 Ownership access not being checked of the stake index in Claim Rewards Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The claimRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.

Locations:

function claimRewards : Line no = 367 - 383

Remediation:

- Ownership Verification: Implement checks to ensure that msg.sender is the owner of the stake identified by _stakeIndex before allowing reward claims.

Impact:

- This function's lack of ownership verification so without verifying ownership , potentially leading to unauthorized reward claims.
- Unauthorized Claim: Any caller can execute claimRewards if they know _stakeIndex, irrespective of stake ownership, leading to unauthorized withdrawal of rewards.

Code Snippet:

```
        function claimRewards(uint256 _stakeIndex) public {
        require(
            poolStakes[_stakeIndex].status !=
                StakeCommons.StakeStatus.WITHDRAWN,
            "Stake has already been withdrawn"
        );
        IToken lodaToken = IToken(ciceroTokenAddress);
        uint256 rewards = computeRewardsForStake(_stakeIndex);

        require(
            lodaToken.balanceOf(address(this)) >= rewards,
            "Insufficient LODA Token"
        );

        lodaToken.transfer(msg.sender, rewards);
        poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards;
    }
```

Reference:

Proof of Vulnerability:

N.A.

4.15 Ownership access not being checked of the stake index in claimAccruedRewards Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The claimAccruedRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.

Locations:

function claimAccuredRewards: Line no = 448 - 464

Remediation:

- Ownership Verification: Implement checks to ensure that msg.sender is the owner of the stake identified by _stakeIndex before allowing reward claims.

Impact:

Impact:

- This function's lack of ownership verification so without verifying ownership , potentially leading to unauthorized reward claims.

Attack:

- **Unauthorized Claim:** Any caller can execute claimRewards if they know _stakeIndex, irrespective of stake ownership, leading to unauthorized withdrawal of rewards.
- **Stake Status Integrity:** Failure to update lastClaimedAt may misrepresent the last claimed time which isn’t correct.

Code Snippet:

```
        function claimAccruedRewards(uint256 _stakeIndex) public {

        require(

            poolStakes[_stakeIndex].status == StakeCommons.StakeStatus.STAKED,

            "Already Withdrawn"

        );



        IToken lodaToken = IToken(ciceroTokenAddress);

        //credit back earned Loda Tokens - how much

        uint256 totalRewardClaimed = computeRewardsForStake(_stakeIndex);

        require(

            lodaToken.balanceOf(address(this)) >= totalRewardClaimed,

            "Insufficient loda balance in the pool"

        );

        lodaToken.transfer(msg.sender, totalRewardClaimed);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += totalRewardClaimed;

        emit RewardDisbursed(msg.sender, totalRewardClaimed, _stakeIndex);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.16 Ownership access not being checked of the stake index in withdraw Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The withdraw function allows users to withdraw a specified amount of USDC from their stake. However, there are critical security and logical issues in the function that need to be addressed to ensure safe and correct operation.

Locations:

function withdraw: Line no = 344 - 387

Remediation:

- **Ownership Verification:** Implement checks to ensure that msg.sender is the owner of the stake identified by _stakeIndex before allowing withdraw.

Impact:

Impact:

- This function's lack of ownership verification so without verifying ownership , potentially leading to unauthorized withdraw.
- **Unauthorized Access:** Anyone can withdraw funds from any stake if they know the _stakeId, leading to potential loss of funds for investors.

Code Snippet:

```
function withdraw(uint256 _amount, uint256 _stakeId) public {

uint256 allowedWithdrawable = poolStakes[_stakeId].amountWithdrawable;

uint256 alreadyWithdrawn = poolStakes[_stakeId].amountAlreadyWithdrawn;

uint256 amountWithdrawable = poolStakes[_stakeId].amountWithdrawable;
```

```
        require(
            _amount <= allowedWithdrawable,
            "Amount is not allowed to withdraw"
        );


        IToken usdcToken = IToken(usdcTokenAddress);
        usdcToken.transfer(msg.sender, _amount);
        poolStakes[_stakeId].amountAlreadyWithdrawn =
            alreadyWithdrawn +
            _amount;
        poolStakes[_stakeId].amountWithdrawable = amountWithdrawable - _amount;


        //to avoid nft burn in case of partial withdrawl
        if (_amount == poolStakes[_stakeId].amountWithdrawable) {
            uint256 noOfInvestments = loanInvestmentCounter[_stakeId].current();
            CiceroNFT nftToken = CiceroNFT(nftTokenAddress);
            if (noOfInvestments == 0) {
                nftToken.burn(_stakeId);
            } else {
                bool completeWithdrawl = true;
                for (uint256 i = 0; i < noOfInvestments; i++) {
                    address loanAddress = loanInvestments[_stakeId][i]
                        .loanAddress;


                    if (
                        Loan(loanAddress).getLoanDetails().loanStatus !=
                        LoanCommon.LoanStatus.COMPLETED
                    ) {
                        completeWithdrawl = false;
                        break;
                    }

                }


                if (completeWithdrawl) {

                    nftToken.burn(_stakeId);

                }

            }

        }

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.17 Ownership access not being checked of the stake index in withdraw Function

Severity	Status	Type of Analysis
🔴 High	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The withdraw function allows users to withdraw a specified amount of USDC from their stake. However, there are critical security and logical issues in the function that need to be addressed to ensure safe and correct operation.

Locations:

function withdraw: Line no = 407 - 435

Remediation:

- Ownership Verification: Implement checks to ensure that msg.sender is the owner of the stake identified by _stakeIndex before allowing withdraw.

Impact:

Impact:

- This function's lack of ownership verification so without verifying ownership , potentially leading to unauthorized withdraw.
- **Unauthorized Access:** Anyone can withdraw funds from any stake if they know the _stakeId, leading to potential loss of funds for investors.

Code Snippet:

```
        function withdraw(uint256 _stakeIndex) public checkMaturity(_stakeIndex) {
        Stake memory _stake = poolStakes[_stakeIndex];
        uint256 slpAmount = _stake.slpTokens;
        uint256 usdcAmount = slpToUsdc(slpAmount);
        require(
            _stake.status != StakeCommons.StakeStatus.WITHDRAWN,
            "Stake has already been withdrawn"
        );
        require(usdcAmount > 0, "Should be a non zero transfer");
        require(
            hasEnoughLiquidity(usdcAmount),
            "Not enough USDC to withdraw in the pool"
        );
        updateLiquidity(usdcAmount);
        unavailableUSDC -= usdcAmount;
        IToken usdcToken = IToken(usdcTokenAddress);
        IToken slpToken = IToken(slpTokenAddress);
        usdcToken.transfer(msg.sender, usdcAmount);
        slpToken.burn(msg.sender, slpAmount);
        if (
            IToken(ciceroTokenAddress).balanceOf(address(this)) >=
            computeRewardsForStake(_stakeIndex)
        ) {
            claimRewards(_stakeIndex);
        }
        poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN;
        replenishFunds(getTotalUsdc());
        emit USDCWithdrawn(msg.sender, usdcAmount);
    }
```

Reference:

Proof of Vulnerability:

N.A.

4.18 Test Functions Vulnerabilities

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The provided functions updateStakeTime_FOR_TEST_REMOVE, addUSDC_FOR_TEST_REMOVE, and removeUSDC_FOR_TEST_REMOVE are intended for testing purposes but pose potential vulnerabilities to the protocol's integrity and security.

Locations:

function updateStakeTime_FOR_TEST_REMOVE: Line no = 486 - 496

function addUSDC_FOR_TEST_REMOVE: Line no = 498 - 500

function removeUSDC_FOR_TEST_REMOVE: Line no = 502- 504

Remediation:

- Secure Testing Environment: Ensure that test functions are disabled or removed from deployed contracts in production environments to prevent unintended access or exploitation if not add Access Control: Restrict access to these functions using appropriate access modifiers (onlyOwner).

Impact:

Impact:

- These functions allow direct manipulation of critical state variables (poolStakes, availableUSDC) without proper access control or validation checks. This can lead to unauthorised modifications or injections of bogus data into the protocol.

Attack:

- **Data Manipulation:** Malicious actors or even unintentional errors during testing could alter critical stake data (stakedAt, lastClaimedAt) in poolStakes, potentially disrupting staking mechanisms or affecting reward calculations.
- **Protocol Inconsistency:** Direct addition or subtraction of availableUSDC without safeguards could lead

to incorrect accounting, affecting the overall availability and balance calculations within the protocol.

- **Exploitation:** If these functions are left exposed or improperly secured, attackers could exploit them to drain funds (availableUSDC) or manipulate stake durations (stakedAt, lastClaimedAt) to their advantage.

Code Snippet:

```
        function updateStakeTime_FOR_TEST_REMOVE(

        uint256 _stakeIndex,

        uint256 _stakedAt

    ) external {

        uint256 lastClaimDiff = poolStakes[_stakeIndex].lastClaimedAt -

            poolStakes[_stakeIndex].stakedAt;

        poolStakes[_stakeIndex].stakedAt = _stakedAt;

        poolStakes[_stakeIndex].lastClaimedAt =

            _stakedAt +

            (lastClaimDiff > 0 ? lastClaimDiff : 0);

    }


function addUSDC_FOR_TEST_REMOVE(uint256 amount) external {

        availableUSDC += amount;

    }


function removeUSDC_FOR_TEST_REMOVE(uint256 amount) external {

        availableUSDC -= amount;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.19 Unauthorized Access to addBorrower Function

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaACL.sol

Description:

The addBorrower function is responsible for adding a new borrower to the protocol. This function checks that the provided borrower address is not zero and that the address is not already a borrower before adding it to the list of borrowers.

Locations:

funciton addBorrower: Line no = 24 - 34

Remediation:

- To prevent unauthorized access to the addBorrower function, implement access control to ensure that only the owner or authorized administrators can add new borrowers.
  Implement the function correctly and securely.

Impact:

- Currently, the addBorrower function is public and can be called by anyone. This lack of access control allows unauthorized users to add themselves or others as borrowers. Given that the protocol issues loans to institution-level borrowers, unauthorized additions can lead to security risks.

Code Snippet:

```
        function addBorrower(address _borrowerAddress)

        public

        checkZeroAddress(_borrowerAddress)

    {

        require(

            borrowers[_borrowerAddress] == false,

            "Address is already a borrower"

        );

        borrowers[_borrowerAddress] = true;

        emit BorrowerAdded(_borrowerAddress);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.20 Unbounded Loops In transferUsdcToLoanAddress function leading to potential DOS attack

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The transferUsdcToLoanAddress function facilitates transferring USDC tokens to the borrower and decreases the availableUSDC and increases the amount for loans forUSDC.

The transferUsdcToLoanAddress function contains nested loops which can lead to a potential denial of service (DOS) attack if the number of allowed stakes is very large. The complexity of the function increases significantly with the number of stakes, potentially causing high gas consumption and transaction failure.

Locations:

function transferUsdcToLoanAddress: Line no = 186-238

function getAllowedStakesAndCount: line no = 453 - 477

Remediation:

- To prevent the DOS attack and improve the function's efficiency, consider optimizing the logic to minimize the number of iterations.

Impact:

Impact

- **High Gas Costs:** The function might consume excessive gas, leading to failure in execution or high costs for the user.
- **Denial of Service:** The function could potentially block the user or the system from executing other transactions due to high resource consumption.

Attack:

- **Potential DOS Attack:** If the number of stakes is very large, the function might consume excessive gas, leading to transaction failure.

Code Snippet:

```
        function transferUsdcToLoanAddress(
    address _loanAddress,
    uint256 _amount,
    uint256 _maturityDate
) public checkLoanAddress returns (uint256[] memory, uint256[] memory) {
    require(_amount <= availableUSDC, "Required funds not available");

    (
        uint256[] memory allowedStakes,
        uint256[] memory allowedStakeData
    ) = getAllowedStakesAndCount(_maturityDate);

    require(
        allowedStakeData[0] >= _amount,
        "insufficient amount available"
    );

    uint256[] memory usedStakeId = new uint256[](allowedStakeData[1]);
    uint256[] memory usedStakeShare = new uint256[](allowedStakeData[1]);
    uint256 totalAmount = 0;

    for (uint256 i = 0; i < allowedStakeData[1]; i++) {
        uint256 stakeId = allowedStakes[i];
        Stake storage stakeForLoanShare = poolStakes[stakeId];
        uint256 amountShare = stakeForLoanShare.amount *
            (stakeForLoanShare.amount / allowedStakeData[0]);

        if (_amount - totalAmount < amountShare) {
                amountShare = _amount - totalAmount;

            }


            stakeForLoanShare.amountWithdrawable -= amountShare;

            loanInvestments[stakeId][

                loanInvestmentCounter[stakeId].current()

            ] = LoanInvestment({

                loanAddress: _loanAddress,

                investedAmount: amountShare

            });

            usedStakeId[i] = stakeId;

            usedStakeShare[i] = amountShare;


            loanInvestmentCounter[stakeId].increment();

            totalAmount += amountShare;


            if (totalAmount == _amount) {

                break;

            }

        }

        IToken usdcToken = IToken(usdcTokenAddress);

        usdcToken.transfer(_loanAddress, _amount);

        availableUSDC -= _amount;

        return (usedStakeId, usedStakeShare);

    }



//NESTED FOR LOOP FROM BELOW FUNCTION


function getAllowedStakesAndCount(uint256 _maturityDate)
```

```
        private

        view

        returns (uint256[] memory, uint256[] memory)

    {

        uint256[] memory allowedStakes = new uint256[](

            poolStakeCounter.current()

        );

        uint256[] memory allowedStakeData = new uint256[](2);


        uint256 allowedStakesCount;

        uint256 allowableAmount = 0;


        for (uint256 i = 0; i < poolStakeCounter.current(); i++) {

            if (poolStakes[i].maturityDate >= _maturityDate) {

                allowableAmount += poolStakes[i].amountWithdrawable;

                allowedStakes[allowedStakesCount] = i;

                allowedStakesCount++;

            }

        }

        allowedStakeData[0] = allowableAmount;

        allowedStakeData[1] = allowedStakesCount;


        return (allowedStakes, allowedStakeData);

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-21-unbounded-loops

Proof of Vulnerability:

N.A.

4.21 updateStakeTime_FOR_TEST_REMOVE Test Function Remove

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The updateStakeTime_FOR_TEST_REMOVE function is designed to update the stake time for testing purposes. However, this function introduces potential security vulnerabilities and should be removed or secured before deploying to production.

Locations:

function updateStakeTime_FOR_TEST_REMOVE: Line no = 612 - 621

Remediation:

- If this function is purely for testing and not intended for production, it should be removed entirely from the deployed contract.

Impact:

- The function allows any users to update the stake time, which can lead to manipulation of staking rewards and other time-dependent calculations.

Code Snippet:

```
        function updateStakeTime_FOR_TEST_REMOVE(uint256 _stakeIndex, uint256 _time)

        external

    {

        uint256 lastClaimDiff = poolStakes[_stakeIndex].lastClaimedAt -

            poolStakes[_stakeIndex].stakedAt;

        poolStakes[_stakeIndex].stakedAt = _time;

        poolStakes[_stakeIndex].lastClaimedAt =

            _time +

            (lastClaimDiff > 0 ? lastClaimDiff : 0);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.22 Differentiation between availableUSDC and unavailableUSDC in withdraw Function is missing

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The withdraw function facilitates the withdrawal of USDC that a user has staked, along with burning corresponding SLP tokens. It also triggers reward claims if available. the protocol is simply converting illiquid funds into liquid funds.

Locations:

function withdraw: Line no = 407-435

Remediation:

- The protocol needs to define clear role for availableUSDC and unavailableUSDC and when the illiquid assets could be used.

Impact:

Impact:

- Protocol is simply making the availableUSDC(Used for loan) convert into unavailableUSDC(It is liquid funds, these are used when investor wants to withdraw stake).

Attack:

- As the role availableUSDC and unavailableUSDC is not clearly defined, in the case of default when funds are in shortage then user can simply draining illiquid funds.

Code Snippet:

```
    function withdraw(uint256 _stakeIndex) public checkMaturity(_stakeIndex) {

    Stake memory _stake = poolStakes[_stakeIndex];

    uint256 slpAmount = _stake.slpTokens;

    uint256 usdcAmount = slpToUsdc(slpAmount);

    require(

        _stake.status != StakeCommons.StakeStatus.WITHDRAWN,

        "Stake has already been withdrawn"

    );

    require(usdcAmount > 0, "Should be a non zero transfer");

    require(

        hasEnoughLiquidity(usdcAmount),

        "Not enough USDC to withdraw in the pool"

    );

    updateLiquidity(usdcAmount);

    unavailableUSDC -= usdcAmount;

    IToken usdcToken = IToken(usdcTokenAddress);

    IToken slpToken = IToken(slpTokenAddress);

    usdcToken.transfer(msg.sender, usdcAmount);

    slpToken.burn(msg.sender, slpAmount);

    if (

        IToken(ciceroTokenAddress).balanceOf(address(this)) >=

        computeRewardsForStake(_stakeIndex)

    ) {

        claimRewards(_stakeIndex);

    }

    poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN;

    replenishFunds(getTotalUsdc());

    emit USDCWithdrawn(msg.sender, usdcAmount);

}
```

Reference:

Proof of Vulnerability:

N.A.

4.23 Divide before multiply

Severity	Status	Type of Analysis
● Medium	Identified	Static

Contract Name:

LodaCompute.sol, Loan.sol, LodaJLP.sol

Description:

Solidity's integer division truncates. Thus, performing division before multiplication can lead to precision loss.

Locations:

LodaCompute.sol: 79-81, 85-86, 82-83.
Loan.sol: 38-54.
LodaJLP.sol: 210-211, 439-45.

Remediation:

Consider ordering multiplication before division.

Impact:

Performing division before multiplication can lead to truncation and significant precision loss in the result.

Code Snippet:

LodaCompute.sol:

```
uint256 rewardInUnitLoda = (timeElapsed * (totalUnitSlpPurchased / 10**18) * perSecondSlpRewardInUnitLoda);


uint256 totalProfitInUnitUsdc = (rewardInUnitUsdc * 3600 * 24 * 365) + profitInUnitUsdc;


uint256 rewardInUnitUsdc = (rewardInUnitLoda * priceOfLodaInUnitUsdc) / 10**18;
```

Loan.sol:

```
loanData = LoanCommon.LoanDetails({
```

```
instituitionName: _instituitionName,

borrower: _borrower,

loanAmount: _loanAmount,

totalRepayment: _monthlyEmi * (_tenure / 30),

outstandingDebt: _monthlyEmi * (_tenure / 30),

tenure: _tenure,

maturityDate: 0,

monthlyEmi: _monthlyEmi,

repaymentDurationInMonths: _tenure / 30,

loanId: _loanId,

interestRate: _interestRate,

loanStatus: LoanCommon.LoanStatus.REQUESTED,

loanApplyDate: block.timestamp,

loanStartDate: 0,

nextDueDate: 0

});
```

LodaJLP.sol:

```
uint256 amountShare = stakeForLoanShare.amount *

(stakeForLoanShare.amount / allowedStakeData[0]);


function investorJlpShare(address _investor) public view returns (uint256) {

uint256 investorStakeCount = investorStakeCounter[_investor].current();

uint256 investorStakeValue;

for (uint256 i = 0; i < investorStakeCount; i++) {

uint256 stakeId = investorStakes[_investor][i];

investorStakeValue =

investorStakeValue +

poolStakes[stakeId].amount;

}
uint256 jlpPercent = (investorStakeValue / availableUSDC) * 100;

return jlpPercent;

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

Proof of Vulnerability:

N.A.

4.24 Incomplete and Vulnerable Default Payments Function

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The defaultPayments function facilitates transferring USDC tokens to the contract and increments availableUSDC without proper checks or handling of the transferred amount in relation to user stakes. However, it lacks essential validations and could pose risks if not appropriately managed.

Locations:

function defaultPayments: Line no = 455 - 463

Remediation:

- **Access Control:** Restrict access to defaultPayments using appropriate access to investors to prevent unauthorized usage and updating their stake in the protocol.
- **Integration with User Stakes:** If the intention is to add amount to a user's stake, modify the function to interact with the appropriate stake management mechanism.

Impact:

Impact:

- This function allows any caller to transfer USDC tokens to the contract and increase availableUSDC without sufficient validation or integration with user stakes. This can lead to unintended consequences such as incorrect accounting of funds or potential manipulation of protocol balances.

Attack:

- Unrestricted Access: Any user can call defaultPayments, potentially leading to unauthorized transfers of USDC tokens to the contract.
- Unregistered user cannot withdraw the amount once he has submitted.

Code Snippet:

```
        function defaultPayments(uint256 amount) public {

        require(amount > 0, "Amount is not valid");

        IToken(usdcTokenAddress).transferFrom(

            msg.sender,

            address(this),

            amount

        );

        availableUSDC += amount;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.25 Investor Stake Limits Not Enforced in stake Function

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The **stake** function is intended to allow JLP investors to stake USDC in the JLP pool. According to the provided documentation, there should be restrictions on the amount a JLP investor can stake:

1. **Initial Stake Limit:** A new JLP investor can only invest up to 5% of the pool size.
2. **Total Stake Limit:** The upper limit per investor over time is 15%.

However, the function does not currently enforce these limits.

Locations:

function stake: Line no = 139 - 168

Remediation:

Enforce Initial Stake Limit:

- Before allowing the staking, calculate the current pool size and ensure the new stake does not exceed 5% of the pool size for new JLP investor.

Enforce Total Stake Limit:

- Track the total amount staked by each investor over time and ensure it does not exceed 15% of the pool size.

Impact:

- As such smart contract doesn't show any vulnerability but as it was mentioned clearly in the docs, that only 5% could be invested initially and 15 % overtime by a JLP user.

Code Snippet:

```
function stake(uint256 _usdcStake, uint256 _tenure)

public
```

```
        checkTenure(_tenure)
    {
        require(_usdcStake > 0, "Should be a non zero transfer");
        uint256 currentStakeId = poolStakeCounter.current();
        IToken usdcToken = IToken(usdcTokenAddress);
        usdcToken.transferFrom(msg.sender, address(this), _usdcStake);
        poolStakes[currentStakeId] = Stake(
            _usdcStake,
            _usdcStake,
            msg.sender,
            block.timestamp,
            block.timestamp + _tenure,
            block.timestamp,
            StakeCommons.StakeStatus.STAKED,
            0,
            0,
            rewardRate[_tenure]
        );

        investorStakes[msg.sender][
            investorStakeCounter[msg.sender].current()
        ] = poolStakeCounter.current();
        availableUSDC = availableUSDC + _usdcStake;
        CiceroNFT nftToken = CiceroNFT(nftTokenAddress);
        nftToken.safeMint(msg.sender, currentStakeId);
        poolStakeCounter.increment();
        investorStakeCounter[msg.sender].increment();
    }
```

Reference:

Proof of Vulnerability:

N.A.

4.26 Missing Update of lastClaimedAt in claimAccruedRewards Function

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The claimAccruedRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.

Locations:

function claimAccruedRewards: Line no = 448 - 464

Remediation:

- Update Stake Status: Update poolStakes[_stakeIndex].lastClaimedAt to reflect the current timestamp when rewards are claimed to maintain accurate stake status.

Impact:

Impact:

- The claimAccruedRewards function fails to update the lastClaimedAt timestamp in poolStakes[_stakeIndex] when rewards are claimed, potentially leading to inaccurate stake data.
- **Misleading Information:** Users relying on lastClaimedAt for stake management or monitoring may receive misleading information, impacting their decisions.

Attack:

- **Inaccurate Stake Status:** Without updating lastClaimedAt, subsequent operations relying on this may produce incorrect results.

Code Snippet:

```
        function claimAccruedRewards(uint256 _stakeIndex) public {

        require(

            poolStakes[_stakeIndex].status == StakeCommons.StakeStatus.STAKED,

            "Already Withdrawn"

        );



        IToken lodaToken = IToken(ciceroTokenAddress);

        //credit back earned Loda Tokens - how much

        uint256 totalRewardClaimed = computeRewardsForStake(_stakeIndex);

        require(

            lodaToken.balanceOf(address(this)) >= totalRewardClaimed,

            "Insufficient loda balance in the pool"

        );

        lodaToken.transfer(msg.sender, totalRewardClaimed);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += totalRewardClaimed;

        emit RewardDisbursed(msg.sender, totalRewardClaimed, _stakeIndex);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.27 Missing Update of lastClaimedAt in claimRewards Function

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The claimRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.

Locations:

function claimRewards: Line no = 303 - 318

Remediation:

- Update Stake Status: Update poolStakes[_stakeIndex].lastClaimedAt to reflect the current timestamp when rewards are claimed to maintain accurate stake status.

Impact:

Impact:

- The claimRewards function fails to update the lastClaimedAt timestamp in poolStakes[_stakeIndex] when rewards are claimed, potentially leading to inaccurate stake status and incorrect reward calculations.
- **Misleading Information:** Users relying on lastClaimedAt for stake management or monitoring may receive misleading information, impacting their decisions.

Attack:

- **Inaccurate Stake Status:** Without updating lastClaimedAt, subsequent operations relying on this may produce incorrect results.

Code Snippet:

```
        function claimRewards(uint256 _stakeIndex) public {

        require(

            canClaimRewards(_stakeIndex),

            "Can't withdraw this stake withdraw"

        );

        IToken lodaToken = IToken(ciceroTokenAddress);

        uint256 rewards = computeRewardsForStake(_stakeIndex);


        require(

            lodaToken.balanceOf(address(this)) >= rewards,

            "Insufficient LODA Token"

        );


        lodaToken.transfer(msg.sender, rewards);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.28 Missing Update of lastClaimedAt in claimRewards Function

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The claimRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.

Locations:

function claimRewards : Line no = 367 - 383

Remediation:

- Update Stake Status: Update poolStakes[_stakeIndex].lastClaimedAt to reflect the current timestamp when rewards are claimed to maintain accurate stake status.

Impact:

- The claimRewards function fails to update the lastClaimedAt timestamp in poolStakes[_stakeIndex] when rewards are claimed, potentially leading to inaccurate stake status and incorrect reward calculations.
- Misleading Information: Users relying on lastClaimedAt for stake management or monitoring may receive misleading information, impacting their decisions.
- Inaccurate Stake Status: Without updating lastClaimedAt, subsequent operations relying on this may produce incorrect results.

Code Snippet:

```
        function claimRewards(uint256 _stakeIndex) public {

        require(

            poolStakes[_stakeIndex].status !=

                StakeCommons.StakeStatus.WITHDRAWN,

            "Stake has already been withdrawn"

        );

        IToken lodaToken = IToken(ciceroTokenAddress);

        uint256 rewards = computeRewardsForStake(_stakeIndex);


        require(

            lodaToken.balanceOf(address(this)) >= rewards,

            "Insufficient LODA Token"

        );


        lodaToken.transfer(msg.sender, rewards);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.29 Insecure Fund Transfer In claimRewards function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The claimRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function claimRewards: Line no = 303 - 318

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function claimRewards(uint256 _stakeIndex) public {

        require(

            canClaimRewards(_stakeIndex),

            "Can't withdraw this stake withdraw"

        );

        IToken lodaToken = IToken(ciceroTokenAddress);

        uint256 rewards = computeRewardsForStake(_stakeIndex);


        require(

            lodaToken.balanceOf(address(this)) >= rewards,

            "Insufficient LODA Token"

        );


        lodaToken.transfer(msg.sender, rewards);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards;

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.30 Insecure Fund Transfer In claimRewards function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The claimRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function claimRewards: Line no = 303 -318

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function claimRewards(uint256 _stakeIndex) public {

        require(

            canClaimRewards(_stakeIndex),

            "Can't withdraw this stake withdraw"

        );

        IToken lodaToken = IToken(ciceroTokenAddress);

        uint256 rewards = computeRewardsForStake(_stakeIndex);


        require(

            lodaToken.balanceOf(address(this)) >= rewards,

            "Insufficient LODA Token"

        );


        lodaToken.transfer(msg.sender, rewards);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards;

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.31 Error handling in slpToUsdc Function should be done

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The slpToUsdc function converts SLP tokens to USDC token based on certain conditions.

Locations:

function slpToUsdc: Line no = 292 - 305

Remediation:

- Add a check for initialStakeRatio should not be equal to 0.

Impact:

Impact: This vulnerability impacts the reliability and integrity of token conversion operations within the protocol:

- **Divide by Zero Risk:** If initialStakeRatio is zero by any chance, division by zero can occur, leading to a unexpected behavior.
- **Error Handling:** Reverting on division by zero without providing clear error messages or handling mechanisms can disrupt user experience.

Code Snippet:

```
        function slpToUsdc(uint256 _slpAmount) public view returns (uint256) {

        uint256 totalSlpSupply = IToken(slpTokenAddress).totalSupply();

        if (totalSlpSupply == 0 || getTotalUsdc() == 0) {

            return _slpAmount / initialStakeRatio;

        } else {

            SLPCompute slpCompute = SLPCompute(slpComputeAddress);

            return

                slpCompute.SLPtoUSDC(

                    _slpAmount,

                    totalSlpSupply,

                    getTotalUsdc()

                );

        }

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.32 Insecure Fund Transfer and Mint Operations In Stake function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

This function stake allows users to stake Loda in the ILP pool and receive LodeStar tokens in return. It also updates various state variables and mappings related to stakes.

However, it does not use the safeTransfer function from OpenZeppelin's SafeERC20 library, nor does it check for successful transfer and minting.

Locations:

function stake: Line no = 170-200

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- If the transferFrom and mint operations fail but are not checked for success, the contract state might not be updated correctly. This could lead to inconsistencies in user balances and stakes, affecting the overall functioning of the contract.
- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function stake(uint256 _stake, uint256 _tenure)

        public

        checkTenure(_tenure)

        checkZeroAddress(ciceroTokenAddress)

    {

        require(_stake > 0, "should be a non zero transfer");

        IToken lodaToken = IToken(ciceroTokenAddress);

        uint256 lodeStarAmount = getLodatoLodestar(_stake);

        lodaToken.transferFrom(msg.sender, address(this), _stake);

        IToken(atticusTokenAddress).mint(msg.sender, lodeStarAmount);

        totalCiceroStaked = totalCiceroStaked + _stake;

        uint256 currentStakeId = poolStakeCounter.current();

        poolStakes[currentStakeId] = Stake(

            _stake,

            lodeStarAmount,

            msg.sender,

            block.timestamp,

            block.timestamp + _tenure,

            block.timestamp,

            COOL_OFF_NONE,

            ILPCommon.StakedTokenType.LODA,

            StakeCommons.StakeStatus.STAKED,

            0,

            rewardRate[_tenure]

        );
        investorStakes[msg.sender][

            investorStakeCounter[msg.sender].current()

        ] = poolStakeCounter.current();

        poolStakeCounter.increment();

        investorStakeCounter[msg.sender].increment();

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.33 Insecure Fund Transfer and Mint Operations In Stake function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

This function stake allows users to stake USDC in the SLP pool and receive SLP tokens in return. It also updates various state variables and mappings related to stakes.

However, it does not use the safeTransfer function from OpenZeppelin's SafeERC20 library, nor does it check for successful transfer and minting.

Locations:

function stake: Line no = 192-230

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

- If the transferFrom and mint operations fail but are not checked for success, the contract state might not be updated correctly. This could lead to inconsistencies in user balances and stakes, affecting the overall functioning of the contract.
- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.
- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function stake(uint256 _usdcStake, uint256 _tenure)

        public

        checkTenure(_tenure)

    {

        require(_usdcStake > 0, "Should be a non zero transfer");

        IToken usdcToken = IToken(usdcTokenAddress);

        uint256 allowance = usdcToken.allowance(msg.sender, address(this));

        require(

            allowance >= _usdcStake,

            "Stake amount should be less or equal to allowance"

        );

        uint256 currentStakeId = poolStakeCounter.current();

        uint256 slpAmount = usdcToSlp(_usdcStake);

        usdcToken.transferFrom(msg.sender, address(this), _usdcStake);

        IToken(slpTokenAddress).mint(msg.sender, slpAmount);

        (uint256 toBeUnavailable, uint256 toBeAvailable) = SLPCompute(

            slpComputeAddress

        ).getUsdcDistribution(getTotalUsdc() + _usdcStake);

        availableUSDC = toBeAvailable;

        unavailableUSDC = toBeUnavailable;

        poolStakes[currentStakeId] = Stake(

            _usdcStake,

            slpAmount,

            msg.sender,

            block.timestamp,

            _tenure,

            block.timestamp,

            StakeCommons.StakeStatus.STAKED,
            0,

            rewardRate[_tenure]

        );


        investorStakes[msg.sender][

            investorStakeCounter[msg.sender].current()

        ] = poolStakeCounter.current();

        poolStakeCounter.increment();

        investorStakeCounter[msg.sender].increment();

        emit USDCStaked(msg.sender, _usdcStake);

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer/understanding-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.34 Insecure Fund Transfer and Mint Operations In StakeUSDC function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

This function stake allows users to stake USDC in the ILP pool and receive LodeStar tokens in return. It also updates various state variables and mappings related to stakes.

However, it does not use the safeTransfer function from OpenZeppelin's SafeERC20 library, nor does it check for successful transfer and minting.

Locations:

function stakeUsdc: Line no = 202 - 233

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- If the transferFrom and mint operations fail but are not checked for success, the contract state might not be updated correctly. This could lead to inconsistencies in user balances and stakes, affecting the overall functioning of the contract.
- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function stakeUsdc(uint256 _usdcAmount, uint256 _tenure)

        public

        checkTenure(_tenure)

    {

        require(_usdcAmount > 0, "should be a non zero transfer");



        IToken usdcToken = IToken(usdcTokenAddress);

        uint256 lodaTokens = usdcToLoda(_usdcAmount);

        usdcToken.transferFrom(msg.sender, address(this), _usdcAmount);

        uint256 lodestarTokens = getLodatoLodestar(lodaTokens);

        IToken(atticusTokenAddress).mint(msg.sender, lodestarTokens);

        uint256 currentStakeId = poolStakeCounter.current();

        poolStakes[currentStakeId] = Stake(

            _usdcAmount,

            lodestarTokens,

            msg.sender,

            block.timestamp,

            block.timestamp + _tenure,

            block.timestamp,

            COOL_OFF_NONE,

            ILPCommon.StakedTokenType.USDC,

            StakeCommons.StakeStatus.STAKED,

            0,

            rewardRate[_tenure]

        );
        usdcInPool += _usdcAmount;

        investorStakes[msg.sender][

            investorStakeCounter[msg.sender].current()

        ] = poolStakeCounter.current();

        poolStakeCounter.increment();

        investorStakeCounter[msg.sender].increment();

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.35 Insecure Fund Transfer In claimAccruedRewards function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The claimAccruedRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function claimAccruedRewards: 448 - 464

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function claimAccruedRewards(uint256 _stakeIndex) public {

        require(

            poolStakes[_stakeIndex].status == StakeCommons.StakeStatus.STAKED,

            "Already Withdrawn"

        );



        IToken lodaToken = IToken(ciceroTokenAddress);

        //credit back earned Loda Tokens - how much

        uint256 totalRewardClaimed = computeRewardsForStake(_stakeIndex);

        require(

            lodaToken.balanceOf(address(this)) >= totalRewardClaimed,

            "Insufficient loda balance in the pool"

        );

        lodaToken.transfer(msg.sender, totalRewardClaimed);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += totalRewardClaimed;

        emit RewardDisbursed(msg.sender, totalRewardClaimed, _stakeIndex);

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.36 Insecure Fund Transfer In claimRewards function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The withdraw function allows any investor to withdraw amount mentioned by the user.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function withdraw: Line no: 407 - 435

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
    function withdraw(uint256 _stakeIndex) public checkMaturity(_stakeIndex) {

    Stake memory _stake = poolStakes[_stakeIndex];

    uint256 slpAmount = _stake.slpTokens;

    uint256 usdcAmount = slpToUsdc(slpAmount);

    require(

        _stake.status != StakeCommons.StakeStatus.WITHDRAWN,

        "Stake has already been withdrawn"

    );

    require(usdcAmount > 0, "Should be a non zero transfer");

    require(

        hasEnoughLiquidity(usdcAmount),

        "Not enough USDC to withdraw in the pool"

    );

    updateLiquidity(usdcAmount);

    unavailableUSDC -= usdcAmount;

    IToken usdcToken = IToken(usdcTokenAddress);

    IToken slpToken = IToken(slpTokenAddress);

    usdcToken.transfer(msg.sender, usdcAmount);

    slpToken.burn(msg.sender, slpAmount);

    if (

        IToken(ciceroTokenAddress).balanceOf(address(this)) >=

        computeRewardsForStake(_stakeIndex)

    ) {

        claimRewards(_stakeIndex);

    }

    poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN;

    replenishFunds(getTotalUsdc());

    emit USDCWithdrawn(msg.sender, usdcAmount);

}
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.37 Insecure Fund Transfer In claimRewards function

Severity	**Status**	**Type of Analysis**
● Low	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The claimRewards function allows any caller to claim rewards for a stake identified by _stakeIndex.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function claimRewards : Line no = 367 - 383

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.
- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function claimRewards(uint256 _stakeIndex) public {

        require(

            poolStakes[_stakeIndex].status !=

                StakeCommons.StakeStatus.WITHDRAWN,

            "Stake has already been withdrawn"

        );

        IToken lodaToken = IToken(ciceroTokenAddress);

        uint256 rewards = computeRewardsForStake(_stakeIndex);


        require(

            lodaToken.balanceOf(address(this)) >= rewards,

            "Insufficient LODA Token"

        );


        lodaToken.transfer(msg.sender, rewards);

        poolStakes[_stakeIndex].rewardsAlreadyClaimed += rewards;

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer/understanding-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.38 Insecure Fund Transfer In defaultPayments function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The defaultPayments function facilitates transferring USDC tokens to the contract and increments availableUSDC.

Locations:

function defaultPayments: Line no: 455 - 463

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.
- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function defaultPayments(uint256 amount) public {

        require(amount > 0, "Amount is not valid");

        IToken(usdcTokenAddress).transferFrom(

            msg.sender,

            address(this),

            amount

        );

        availableUSDC += amount;

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.39 Insecure Fund Transfer In stake function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The stake function is intended to allow JLP investors to stake USDC in the JLP pool.

Locations:

function stake: Line no = 139 - 168

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function stake(uint256 _usdcStake, uint256 _tenure)
        public
        checkTenure(_tenure)
    {
        require(_usdcStake > 0, "Should be a non zero transfer");
        uint256 currentStakeId = poolStakeCounter.current();
        IToken usdcToken = IToken(usdcTokenAddress);
        usdcToken.transferFrom(msg.sender, address(this), _usdcStake);
        poolStakes[currentStakeId] = Stake(
            _usdcStake,
            _usdcStake,
            msg.sender,
            block.timestamp,
            block.timestamp + _tenure,
            block.timestamp,
            StakeCommons.StakeStatus.STAKED,
            0,
            0,
            rewardRate[_tenure]
        );

        investorStakes[msg.sender][
            investorStakeCounter[msg.sender].current()
        ] = poolStakeCounter.current();
        availableUSDC = availableUSDC + _usdcStake;
        CiceroNFT nftToken = CiceroNFT(nftTokenAddress);
        nftToken.safeMint(msg.sender, currentStakeId);
        poolStakeCounter.increment();
        investorStakeCounter[msg.sender].increment();
    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.40 Insecure Fund Transfer in transferFundsToPools Function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

loan.sol

Description:

The transferFundsToPools function transfers funds to specified pool addresses. However, it does not use the safeTransfer function from OpenZeppelin's SafeERC20 library, nor does it check for successful transfer.

Locations:

function transferFundsToPools: Line no = 114-121

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function transferFundsToPools(address _to, uint256 _amount)

        public

        checkLoanFactoryAddress

    {

        IToken usdcToken = IToken(usdcTokenAddress);

        usdcToken.transfer(_to, _amount);

    }

}
```

Reference:

Proof of Vulnerability:

N.A.

4.41 Insecure Fund Transfer In transferUsdcToLoanAddress function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The transferUsdcToLoanAddress function facilitates transferring USDC tokens to the borrower and decreases the availableUSDC and increases the amount for loans forUSDC.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function transferUsdcToLoanAddress: Line no = 437 - 453

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.
- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function transferUsdcToLoanAddress(address reciever, uint256 amountToSend)

        public

        checkLoanAddress

    {

        require(amountToSend >= 0, "Amount is smaller than zero");



        require(

            amountToSend <= availableUSDC,

            "No usdc available for asked amount"

        );



        availableUSDC -= amountToSend;

        usdcForLoans += amountToSend;



        IToken(usdcTokenAddress).transfer(reciever, amountToSend);

        emit USDCTransfered(reciever, amountToSend);

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer/understanding-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.42 Insecure Fund Transfer In transferUsdcToLoanAddress function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The transferUsdcToLoanAddress function facilitates transferring USDC tokens to the borrower and decreases the availableUSDC and increases the amount for loans forUSDC.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function transferUsdcToLoanAddress: Line no = 186 - 238

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
            function transferUsdcToLoanAddress(

        address _loanAddress,

        uint256 _amount,

        uint256 _maturityDate

    ) public checkLoanAddress returns (uint256[] memory, uint256[] memory) {

        require(_amount <= availableUSDC, "Required funds not available");


        (

            uint256[] memory allowedStakes,

            uint256[] memory allowedStakeData

        ) = getAllowedStakesAndCount(_maturityDate);


        require(

            allowedStakeData[0] >= _amount,

            "insufficient amount available"

        );


        uint256[] memory usedStakeId = new uint256[](allowedStakeData[1]);

        uint256[] memory usedStakeShare = new uint256[](allowedStakeData[1]);

        uint256 totalAmount = 0;


        for (uint256 i = 0; i < allowedStakeData[1]; i++) {

            uint256 stakeId = allowedStakes[i];

            Stake storage stakeForLoanShare = poolStakes[stakeId];

            uint256 amountShare = stakeForLoanShare.amount *

                (stakeForLoanShare.amount / allowedStakeData[0]);


            if (_amount - totalAmount < amountShare) {
                amountShare = _amount - totalAmount;

            }


            stakeForLoanShare.amountWithdrawable -= amountShare;

            loanInvestments[stakeId][

                loanInvestmentCounter[stakeId].current()

            ] = LoanInvestment({

                loanAddress: _loanAddress,

                investedAmount: amountShare

            });

            usedStakeId[i] = stakeId;

            usedStakeShare[i] = amountShare;


            loanInvestmentCounter[stakeId].increment();

            totalAmount += amountShare;


            if (totalAmount == _amount) {

                break;

            }

        }

        IToken usdcToken = IToken(usdcTokenAddress);

        usdcToken.transfer(_loanAddress, _amount);

        availableUSDC -= _amount;

        return (usedStakeId, usedStakeShare);

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.43 Insecure Fund Transfer In withdraw function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The withdraw function allows any investor to withdraw loda amount from the ILP Pool.But it is not using safeTransfer method and checking whether the transfer was successful or not.

Locations:

function withdraw: Line no = 249 - 281

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.

Attack:

- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
    function withdraw(uint256 _stakeIndex) external {
```

```
function withdraw(uint256 _stakeIndex) external {

Stake memory _stake = poolStakes[_stakeIndex];


require(

    _stake.status == StakeCommons.StakeStatus.STAKED,

    "Already Withdrawn"

);


require(

    _stake.maturityDate <= block.timestamp ||

        (_stake.coolOffStarted != 0 &&

            getCoolOffPeriod(_stakeIndex) + _stake.coolOffStarted <=

            block.timestamp),

    "still within cooling off period"

);

IToken lodaToken = IToken(ciceroTokenAddress);

IToken lodestarToken = IToken(atticusTokenAddress);


uint256 currentLodaAmount = getLodestarToLoda(_stake.atticusTokens);

require(

    currentLodaAmount <= lodaToken.balanceOf(address(this)),

    "Insufficient loda tokens in pool"

);

lodestarToken.burn(msg.sender, _stake.atticusTokens);

lodaToken.transfer(msg.sender, currentLodaAmount);

uint256 claimableRewards = computeRewardsForStake(_stakeIndex);

if (lodaToken.balanceOf(address(this)) >= claimableRewards) {

    lodaToken.transfer(msg.sender, claimableRewards);

    poolStakes[_stakeIndex].rewardsAlreadyClaimed += claimableRewards;

}
```

```
        poolStakes[_stakeIndex].status = StakeCommons.StakeStatus.WITHDRAWN;

    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.44 Insecure Transfer function in makeLoanRepayment Function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

loan.sol

Description:

The makeLoanRepayment function does not check whether the transfer of funds was successful. It also does not use the safeTransfer function from OpenZeppelin's SafeERC20 library, nor does it check for successful transfer.

Locations:

function makeLoanRepayment: Line no = 65-89

Remediation:

The use of OpenZeppelin's SafeERC20 library provides a robust mitigation strategy. The library's safeTransfer() and safeTransferFrom() functions ensure the safe and reliable transfer of tokens, including non-standard compliant ones. They handle the potential discrepancies in function signatures, mitigating the associated risks and ensuring that the transfer operations do not fail.

Impact:

- Non-Standard Compliant Tokens: Non-standard-compliant tokens can cause standard ERC20 functions to revert unexpectedly. The inconsistency in function signatures between the non-standard tokens and the standard ERC20 tokens results in unexpected behaviour, causing disruptions in basic operations like token deposits and withdrawals.
- The use of standard ERC20 transfer functions with non-standard tokens may result in transaction failures or even serious accounting errors in smart contracts. The latter may present severe security vulnerabilities, as this could be exploited by malicious actors to manipulate the contract state to their advantage.

Code Snippet:

```
        function makeLoanRepayment(uint256 _amount, string memory _selection)
        public
        checkLoanFactoryAddress
    {
        IToken usdcToken = IToken(usdcTokenAddress);
        usdcToken.transferFrom(loanData.borrower, address(this), _amount);
        repaymentCounter.increment();

        countToRepayment[repaymentCounter.current()] = LoanCommon.Repayment({
            dueDate: loanData.nextDueDate,
            paymentDate: block.timestamp,
            amount: _amount,
            selection: _selection
        });

        loanData.nextDueDate += (30 days);
        if (
            keccak256(abi.encodePacked(_selection)) ==
            keccak256(abi.encodePacked("totalPayment"))
        ) {
            loanData.outstandingDebt = 0;
        } else {
            loanData.outstandingDebt -= _amount;
        }
    }
```

Reference:

https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-10-erc20-transfer-and-safetransfer/understanding-erc20-transfer-and-safetransfer

Proof of Vulnerability:

N.A.

4.45 Missing events access control

Severity	Status	Type of Analysis
● Low	Identified	Static

Contract Name:

LodaSLP.sol, CiceroNFT.sol

Description:

This check detects instances where critical access control parameters, such as ownership changes, are not accompanied by emitted events. Events are crucial for transparency and off-chain tracking of important state changes within the contract.

Locations:

LodaJLP.sol: 122,163.
CiceroNFT.sol: 41.
LodaSLP.sol:163.

Remediation:

Emit an event for critical parameter changes.

Impact:

Missing events for critical state changes reduce transparency.

Code Snippet:

LodaJLP.sol:

```
function updateLoanFactoryAddress(address _loanFactoryAddress) external onlyOwner { loanFactoryAddress = _l
oanFactoryAddress; }
```

CiceroNFT.sol:

```
function updateLodaJLPAddress(address _lodaJLPAddress) external onlyOwner {

lodaJLPAddress = _lodaJLPAddress;

}
```

LodaSLP.sol:

```
function updateLoanFactoryAddress(address _loanFactoryAddress)

external

onlyOwner

{

loanFactoryAddress = _loanFactoryAddress;

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-access-control

Proof of Vulnerability:

N.A.

4.46 Missing zero address validation

Severity	Status	Type of Analysis
● Low	Identified	Static

Contract Name:

LoanFactory.sol, ILPCompute.sol, Loan.sol, LodaILP.sol, LodaJLP.sol, LodaSLP.sol, CiceroNFT.sol

Description:

Detect missing zero address validation.This check identifies instances where addresses are not validated to ensure they are not set to the zero address (`address(0)`). Failure to validate addresses can lead to unintended behavior or loss of ownership, especially in functions where address assignments or changes are critical.

Locations:

LoanFactory.sol: 42, 49, 56, 66.
ILPCompute.sol: 29.
Loan.sol: 56, 57.
LodaILP.sol: 115, 125,135,163.
LodaJLP.sol: 81, 112, 122, 132.
LodaSLP.sol: 119, 133, 143, 153, 163.
CiceroNFT.sol: 41.

Remediation:

Check that the address is not zero. Always validate addresses to ensure they are not set to the zero address (`address(0)`) before proceeding with critical operations or assignments. This practice mitigates risks associated with unintended address assignments and maintains the integrity and security of the smart contract.

Impact:

Failing to validate addresses against zero can lead to loss of control over critical contract functions or assets. Malicious actors could exploit functions with missing zero address checks to disable or compromise contract operations.

Code Snippet:

LoanFactory.sol:

```
function updateLodaJLPAddress(address _lodaJLPAddress) external onlyOwner {

lodaJLPAddress = _lodaJLPAddress;
}


function updateLodaSLPAddress(address _lodaSLPAddress) external onlyOwner {

lodaSLPAddress = _lodaSLPAddress;

}


function updateLodaILPAddress(address _lodaILPAddress) external onlyOwner {

lodaILPAddress = _lodaILPAddress;

}


function updateUSDCTokenAddress(address _usdcTokenAddress)

external

onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;

}
```

ILPCompute.sol:

```
function updateRealMathAddress(address _realMathAddress)

external

onlyOwner

{

realMathAddress = _realMathAddress;

}
```

Loan.sol:

```
constructor(

string memory _instituitionName,

address _borrower,
uint256 _loanAmount,

uint256 _tenure,

uint256 _loanId,

uint256 _interestRate,

uint256 _monthlyEmi,

address _usdcTokenAddress,

address _loanFactory

) {

initialize();

_disableInitializers();

loanData = LoanCommon.LoanDetails({

instituitionName: _instituitionName,

borrower: _borrower,

loanAmount: _loanAmount,

totalRepayment: _monthlyEmi * (_tenure / 30),

outstandingDebt: _monthlyEmi * (_tenure / 30),

tenure: _tenure,

maturityDate: 0,

monthlyEmi: _monthlyEmi,

repaymentDurationInMonths: _tenure / 30,

loanId: _loanId,

interestRate: _interestRate,

loanStatus: LoanCommon.LoanStatus.REQUESTED,

loanApplyDate: block.timestamp,

loanStartDate: 0,

nextDueDate: 0

});

usdcTokenAddress = _usdcTokenAddress;

loanFactoryAddress = _loanFactory;

}
```

LodaILP.sol:

```
function updateUSDCTokenAddress(address _usdcTokenAddress)

external

onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;

}



function updateAtticusToken(address _atticusTokenAddress)

external

onlyOwner

{

atticusTokenAddress = _atticusTokenAddress;

}



function updateILPComputeAddress(address _ILPComputeAddress)

external

onlyOwner

{

ILPComputeAddress = _ILPComputeAddress;

}



function updateSLPPoolAddress(address _SLPPoolAddress) external onlyOwner {

SLPPoolAddress = _SLPPoolAddress;

}
```

LodaJLP.sol:

```
function updateCiceroTokenAddress(address _ciceroTokenAddress)
```

```
external

onlyOwner

{

ciceroTokenAddress = _ciceroTokenAddress;

}



function updatenftTokenAddress(address _nftTokenAddress)

external

onlyOwner

{

nftTokenAddress = _nftTokenAddress;

}



function updateLoanFactoryAddress(address _loanFactoryAddress)

external

onlyOwner

{

loanFactoryAddress = _loanFactoryAddress;

}



function updateUSDCTokenAddress(address _usdcTokenAddress)

external

onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;

}
```

LodaSLP.sol:

```
function updateUSDCTokenAddress(address _usdcTokenAddress)

external
```

```
onlyOwner

{

usdcTokenAddress = _usdcTokenAddress;

}



function updateSLPTokenAddress(address _slpTokenAddress)

external

onlyOwner

{

slpTokenAddress = _slpTokenAddress;

}



function updateCiceroTokenAddress(address _ciceroTokenAddress)

external

onlyOwner

{

ciceroTokenAddress = _ciceroTokenAddress;

}



function updateSLPComputeAddress(address _slpComputeAddress)

external

onlyOwner

{

slpComputeAddress = _slpComputeAddress;

}



function updateLoanFactoryAddress(address _loanFactoryAddress)

external

onlyOwner

{
```

```
loanFactoryAddress = _loanFactoryAddress;

}
```

CiceroNFT.sol:

```
function updateLodaJLPAddress(address _lodaJLPAddress) external onlyOwner {

lodaJLPAddress = _lodaJLPAddress;

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Proof of Vulnerability:

N.A.

4.47 Simplification of rewardsAccruedPerStake Function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The rewardsAccruedPerStake function calculates accrued rewards for a stake based on its maturity date compared to the current timestamp. It retrieves stake data from poolStakes[_stakeId] and calculates rewards differently based on whether the stake has matured or not.

Locations:

function rewardsAccruedPerStake: Line no = 235 - 254

Remediation:

- Remove the unnecessary condition checks and simplify the rewards calculation logic.

Impact:

- Reduced Gas Usage: Removing the unnecessary condition checks will make the function more gas-efficient. Each conditional statement in Solidity incurs a gas cost, so eliminating them will reduce the overall gas consumption of the function.
- There are no direct attack vulnerabilities introduced by these unnecessary checks.

Code Snippet:

```
        function rewardsAccruedPerStake(uint256 _stakeId)

        public

        view

        returns (uint256)

    {

        Stake memory stakeData = poolStakes[_stakeId];

        uint256 maturityDate = stakeData.stakedAt + stakeData.tenure;


        if (maturityDate >= block.timestamp) {

            return

                stakeData.rewardRatePerSec *

                (block.timestamp - stakeData.stakedAt) *

                stakeData.amount;

        } else {

            return

                stakeData.rewardRatePerSec *

                (maturityDate - stakeData.stakedAt) *

                stakeData.amount;

        }

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.48 Simplification of rewardsAccruedPerStake Function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The rewardsAccruedPerStake function calculates accrued rewards for a stake based on its maturity date compared to the current timestamp. It retrieves stake data from poolStakes[_stakeId] and calculates rewards differently based on whether the stake has matured or not.

Locations:

function rewardsAccruedPerStake: Line no = 252 - 269

Remediation:

- Remove the unnecessary condition checks and simplify the rewards calculation logic as this will improve the gas effciency.

Impact:

Impact:

- **Reduced Gas Usage:** Removing the unnecessary condition checks will make the function more gas-efficient. Each conditional statement in Solidity incurs a gas cost, so eliminating them will reduce the overall gas consumption of the function.

Attack:

1. There are no direct security vulnerabilities introduced by these unnecessary checks.

Code Snippet:

```
        function rewardsAccruedPerStake(uint256 _stakeId)

        public

        view

        returns (uint256)

    {

        Stake memory stakeData = poolStakes[_stakeId];

        if (stakeData.maturityDate >= block.timestamp) {

            return

                stakeData.rewardRatePerSec *

                (block.timestamp - stakeData.stakedAt) *

                stakeData.amount;

        } else {

            return

                stakeData.rewardRatePerSec *

                (stakeData.maturityDate - stakeData.stakedAt) *

                stakeData.amount;

        }

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.49 Unnecessary Condition Checks in allocateProfitsToPools Function

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

loanFactory.sol

Description:

The allocateProfitsToPools function distributes the profits from a loan among various pools and the treasury. It calculates the shares for each pool and the treasury, then transfers these amounts accordingly. However, the function contains unnecessary condition checks that always evaluate to true. Specifically, it checks if the sum of totalIlpProfits, totalJlpProfits, and totalSlpProfits is less than or equal to the total profit and adjusts totalJlpProfits accordingly.

Locations:

function allocateProfitsToPools: Line no = 164-211

Remediation:

- Remove the unnecessary condition checks and simplify the profit distribution logic.

Impact:

- Reduced Gas Usage: Removing the unnecessary condition checks will make the function more gas-efficient. Each conditional statement in Solidity incurs a gas cost, so eliminating them will reduce the overall gas consumption of the function.
  There are no direct security vulnerabilities introduced by this.

Code Snippet:

```
        function allocateProfitsToPools(uint256 _loanId) public {

    address loanAddress = loanIdToAddressMapping[_loanId];

    Loan targetLoan = loanDetails[_loanId];

    LoanCommon.LoanDetails memory loan = targetLoan.getLoanDetails();

    uint256 principal = loan.loanAmount;

    uint256 jlpShare = principal / 5;
```

```
        uint256 slpShare = principal - jlpShare;


        uint256 profit = IToken(usdcTokenAddress).balanceOf(loanAddress) -

            principal;


        uint256 totalTreasuryProfits = uint256((10 * profit) / 100);

        uint256 totalIlpProfits = uint256((20 * profit) / 100);

        uint256 totalSlpProfits = uint256((30 * profit) / 100);

        uint256 totalJlpProfits = uint256((40 * profit) / 100);


        treasury[_loanId] = totalTreasuryProfits;


        if ((totalIlpProfits + totalJlpProfits + totalSlpProfits) <= profit) {

            totalJlpProfits += (profit -

                (totalIlpProfits + totalJlpProfits + totalSlpProfits));

        }

        if ((totalIlpProfits + totalJlpProfits + totalSlpProfits) >= profit) {

            totalJlpProfits -= (profit -

                (totalIlpProfits + totalJlpProfits + totalSlpProfits));

        }


        Loan(loanAddress).transferFundsToPools(lodaILPAddress, totalIlpProfits);

        LodaILP(lodaILPAddress).profitShare(_loanId, totalIlpProfits);


        Loan(loanAddress).transferFundsToPools(

            lodaSLPAddress,

            totalSlpProfits + slpShare

        );

        LodaSLP(lodaSLPAddress).profitShare(
            _loanId,

            totalSlpProfits + slpShare

        );


        Loan(loanAddress).transferFundsToPools(

            lodaJLPAddress,

            totalJlpProfits + jlpShare

        );

        LodaJLP(lodaJLPAddress).profitShare(

            _loanId,

            totalJlpProfits + jlpShare

        );

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.50 Unused return

Severity	Status	Type of Analysis
● Low	Identified	Static

Contract Name:

LoanFactory.sol

Description:

The return value of an external call is not stored in a local or state variable.

Locations:

LoanFactory.sol: 117-121

Remediation:

Ensure that all the return values of the function calls are used.

Impact:

The impact of not storing the return value of an external call in a local or state variable is that the computation performed by the call has no effect, potentially leading to logical errors and unintended behavior in the contract.

Code Snippet:

```
        function getFirstLossCapital(

address _loanAddress,

uint256 _amount,

uint256 _maturityDate

) private {

LodaJLP(lodaJLPAddress).transferUsdcToLoanAddress(

_loanAddress,

_amount,

_maturityDate

);

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return

Proof of Vulnerability:

N.A.

4.51 Custom Error Message Missing in computeRewardsForStake Function

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

LodaILP.sol

Description:

The computeRewardsForStake function calculates the total rewards that can be claimed for a specific stake identified by _stakeId. If the stake cannot currently claim rewards (as determined by canClaimRewards(_stakeId)), the function returns 0 without providing a custom error message.

Locations:

function computeRewardsForStake: Line no = 432- 443

Remediation:

- Add Custom Error Message: Include a descriptive error message in the computeRewardsForStake function to clarify why rewards cannot be computed for a specific stake.

Impact:

Impact:

- Without a custom error message, users interacting with the computeRewardsForStake function may receive unexpected behavior when attempting to calculate rewards for stakes that are not eligible.

Attack:

- It doesn’t have any attack vector associated with it, it improves the error handling process.

Code Snippet:

```
        function computeRewardsForStake(uint256 _stakeId)

        public

        view

        returns (uint256)

    {

        if (!canClaimRewards(_stakeId)) {

            return 0;

        }



        uint256 totalRewards = rewardsAccruedPerStake(_stakeId);

        return (totalRewards - poolStakes[_stakeId].rewardsAlreadyClaimed);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.52 Custom Error Message Missing in computeRewardsForStake Function

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

LodaJLP.sol

Description:

The computeRewardsForStake function calculates the total rewards that can be claimed for a specific stake identified by _stakeId. If the stake cannot currently claim rewards (as determined by canClaimRewards(_stakeId)), the function returns 0 without providing a custom error message.

Locations:

function computeRewardsForStake: Line no = 288 - 299

Remediation:

- Add Custom Error Message: Include a descriptive error message in the computeRewardsForStake function to clarify why rewards cannot be computed for a specific stake.

Impact:

Impact:

- Without a custom error message, users interacting with the computeRewardsForStake function may receive unexpected behavior when attempting to calculate rewards for stakes that are not eligible. This can lead to confusion and difficulty in understanding why rewards are not computed as expected.

Attack:

- It doesn’t have any attack vector associated with it, it improves the error handling process.

Code Snippet:

```
        function computeRewardsForStake(uint256 _stakeId)

public

view

returns (uint256)

{

if (!canClaimRewards(_stakeId)) {

return 0;

}

uint256 totalRewards = rewardsAccruedPerStake(_stakeId);

return (totalRewards - poolStakes[_stakeId].rewardsAlreadyClaimed);

}
```

Reference:

Proof of Vulnerability:

N.A.

4.53 Custom Error Message Missing in computeRewardsForStake Function

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

LodaSLP.sol

Description:

The computeRewardsForStake function calculates the total rewards that can be claimed for a specific stake identified by _stakeId. If the stake cannot currently claim rewards (as determined by canClaimRewards(_stakeId)), the function returns 0 without providing a custom error message.

Locations:

function computeRewardsForStake: Line no = 352 - 363

Remediation:

- **Add Custom Error Message:** Include a descriptive error message in the computeRewardsForStake function to clarify why rewards cannot be computed for a specific stake.

Impact:

- Without a custom error message, users interacting with the computeRewardsForStake function may receive unexpected behavior when attempting to calculate rewards for stakes that are not eligible. This can lead to confusion and difficulty in understanding why rewards are not computed as expected.

- It doesn't have any attack vector associated with it, it improves the error handling process.

Code Snippet:

```
        function computeRewardsForStake(uint256 _stakeId)

        public

        view

        returns (uint256)

    {

        if (!canClaimRewards(_stakeId)) {

            return 0;

        }



        uint256 totalRewards = rewardsAccruedPerStake(_stakeId);

        return (totalRewards - poolStakes[_stakeId].rewardsAlreadyClaimed);

    }
```

Reference:

Proof of Vulnerability:

N.A.

4.54 Dead-code

Severity	Status	Type of Analysis
● Informational	Identified	Static

Contract Name:

LoanFactory.sol

Description:

Functions that are not used.

Locations:

LoanFactory.sol: 107-110.

Remediation:

Remove unused functions.

Impact:

It doesn't have any direct attack vector associated with it.

Code Snippet:

LoanFactory.sol:

```
function getLoanAddress(uint256 _index) private view returns (address) {

require(_index < loanCount.current(), "index is out of bounds");

return loanIdToAddressMapping[_index];

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Proof of Vulnerability:

N.A.

4.55 Solidity pragma should be specific, not wide

Severity	Status	Type of Analysis
● Informational	Identified	Static

Contract Name:

Loan.sol, LodaJLP.sol, LodaILP.sol, LodaSLP.sol

Description:

Using a wide version in the Solidity pragma statement (^0.8.9) is discouraged. It's recommended to specify a particular version to ensure compatibility and avoid unexpected behavior due to potential breaking changes in future compiler versions.

Locations:

Loan.sol, LodaJLP.sol, LodaILP.sol, LodaSLP.sol

Remediation:

Update the pragma statements in the contracts to specify a particular version of Solidity. For example, replace pragma solidity ^0.8.9; with pragma solidity 0.8.9;.

Impact:

Failure to specify a specific version may lead to compatibility issues or unexpected behavior in future compiler versions. It's important to follow best practices to ensure the stability and security of the contracts.

Code Snippet:

```
        pragma solidity ^0.8.9;
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Proof of Vulnerability:

N.A.

4.56 transferUsdcToLoanAddress contains a tautology

Severity	Status	Type of Analysis
● Informational	Identified	Static

Contract Name:

LodaSLP.sol

Description:

Detects expressions that are tautologies.

Locations:

LodaSLP.sol: Line no = 434-452

Remediation:

Fix the incorrect comparison by changing the value type or the comparison.

Impact:

`x` is a `uint256`, so `x >= 0` will be always true. `y` is a `uint8`, so `y <512` will be always true.

```
contract A {
    function f(uint x) public {
        // ...
        if (x >= 0) { // bad -- always true
           // ...
        }
        // ...
    }

    function g(uint8 y) public returns (bool) {
        // ...
        return (y < 512); // bad!
        // ...
    }
}
```

Code Snippet:

```
        function transferUsdcToLoanAddress(address reciever, uint256 amountToSend)

public

checkLoanAddress

{

require(amountToSend >= 0, "Amount is smaller than zero");

require(

amountToSend <= availableUSDC,

"No usdc available for asked amount"

);

availableUSDC -= amountToSend;

usdcForLoans += amountToSend;

IToken(usdcTokenAddress).transfer(reciever, amountToSend);

emit USDCTransfered(reciever, amountToSend);

}
```

Reference:

https://github.com/crytic/slither/wiki/Detector-Documentation#tautology-or-contradiction

Proof of Vulnerability:

N.A.

5.0 Auditing Approach and Methodologies applied

Throughout the audit of the smart contract, care was taken to ensure:

- Overall quality of code
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Mathematical calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of token standards.
- Efficient use of gas.
- Code is safe from Re-entrancy and other vulnerabilities.

A combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

5.1 Structural Analysis

In this step we have analysed the design patterns and structure of all smart contracts. A thorough check was completed to ensure all Smart contracts are structured in a way that will not result in future problems.

5.2 Static Analysis

Static Analysis of smart contracts was undertaken to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

5.3 Code Review / Manual Analysis

Manual Analysis or review of done to identify new vulnerabilities or to verify the vulnerabilities found during the Static Analysis. The contracts were completely manually analysed, and their logic was checked and compared with the one described in the whitepaper. It should also be noted that the results of the automated analysis were verified manually.

5.4 Gas Consumption

In this step, we checked the behaviour of all smart contracts in production. Checks were completed to understand how much gas gets consumed, along with the possibilities of optimisation of code to reduce gas consumption.

5.5 Tools & Platforms Used For Audit

Slither, Aderyn

5.6 Checked Vulnerabilities

We have scanned Cicero smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC-20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

6.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of Cicero and methods for exploiting them. Entersoft recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while Entersoft considers the major security vulnerabilities of the analyzed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Smart Contract described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities will also change. Entersoft makes no undertaking to supplement or update this report based on changed circumstances or facts of which Entersoft becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that Entersoft use certain software or hardware products manufactured or maintained by other vendors. Entersoft bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, Entersoft does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by Entersoft for the exclusive benefit of Cicero and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between Entersoft and Cicero governs the disclosure of this report to all other parties including product vendors and suppliers.