SMART CONTRACT AUDIT REPORT

for

MAPLE LABS

Prepared By: Shuxiao Wang

PeckShield

May 2, 2021

Document Properties

Client	Maple Labs
Title	Smart Contract Audit Report
Target	Maple
Version	1.0.2
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0.2	May 2, 2021	Xuxian Jiang	Final Release (Amended #2)
1.0.1	April 12, 2021	Xuxian Jiang	Final Release (Amended #1)
1.0	March 19, 2021	Xuxian Jiang	Final Release
1.0-rc1	March 17, 2021	Xuxian Jiang	Release Candidate #1
0.5	March 13, 2021	Xuxian Jiang	Add More Findings #4
0.4	March 12, 2021	Xuxian Jiang	Add More Findings #3
0.3	March 10, 2021	Xuxian Jiang	Add More Findings #2
0.2	March 7, 2021	Xuxian Jiang	Add More Findings #1
0.1	March 1, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Maple** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Maple

`Maple` is a decentralized corporate credit market that aims to provide capital to institutional borrowers through globally accessible fixed-income yield opportunities. In particular, `liquidity pools` are utilized to aggregate funding from `liquidity providers` and are loaned out to earn interest. The pools are professionally managed by `pool delegates` to provide as a sustainable yield source. And `Borrowers` request capital from the `Maple` protocol by performing transparent and efficient financing entirely on-chain. `Pool delegates` perform diligence and agree terms with `Borrowers`. To be qualified, `pool delegates` are required to stake the protocol token, i.e., `MPL`, in their pools to cover defaults, aligning their incentives with `liquidity providers`.

The basic information of Maple is as follows:

Table 1.1: Basic Information of Maple

Item	Description
Issuer	Maple Labs
Website	https://maple.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 2, 2021

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

- https://github.com/maple-labs/maple-core.git (05ef95f)

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

- https://github.com/maple-labs/maple-core.git (d921a7c)

1.2 About PeckShield

PeckShield Inc. [20] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [19]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [18], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Maple protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	3	■■■
Low	8	■■■■■■■■
Informational	5	■■■■■
Total	18	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, 8 low-severity vulnerabilities, and 5 informational recommendations.

Table 2.1: Key Maple Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-002	Medium	Proper Effective Stake Date Calculation	Time and State	Fixed
PVE-003	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-004	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-005	Low	Avoidance Of Zero Amount Transfer	Coding Practices	Fixed
PVE-006	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Fixed
PVE-007	Informational	Better Handling of Privilege Transfers	Security Features	Fixed
PVE-008	High	Possible Front-Running For Reduced Stake Requirements	Time and State	Resolved
PVE-009	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-010	Low	Simplification of PoolLib::updateDepositDate()	Coding Practices	Fixed
PVE-011	Low	Timely updateFundsReceived() in Loan Management	Business Logic	Fixed
PVE-012	Low	Lack Of Proper Enforcement Of fundingPeriodSeconds	Business Logic	Resolved
PVE-013	Informational	Removal of Unused Code	Coding Practices	Fixed
PVE-014	Informational	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Resolved
PVE-015	High	Bypass of lockupPeriod in Pool::withdraw()	Business Logic	Fixed
PVE-016	Informational	Suggested Addition of rescueToken()	Business Logic	Fixed
PVE-017	Medium	Potential Collusion Between PoolDelegate And Borrowers	Business Logic	Resolved
PVE-018	Medium	Revisited Assumption on Trusted Governance	Security Features	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

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

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1126 [2]

Description

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

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

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

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

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

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

Listing 3.1: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `MapleTreasury::convertERC20()` routine as an example. This routine is designed to trigger default handling. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 93): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
85      function convertERC20(address asset) isGovernor public {
86          require(asset != fundsToken, "MapleTreasury:ASSET_EQUALS_FUNDS_TOKEN");

88          IGlobals _globals = IGlobals(globals);

90          uint256 assetBalance = IERC20(asset).balanceOf(address(this));
91          uint256 minAmount    = Util.calcMinAmount(_globals, asset, fundsToken,
                assetBalance);

93          IERC20(asset).approve(uniswapRouter, assetBalance);

95          address uniswapAssetForPath = _globals.defaultUniswapPath(asset, fundsToken);
96          bool middleAsset = uniswapAssetForPath != fundsToken && uniswapAssetForPath !=
                address(0);

98          address[] memory path = new address[](middleAsset ? 3 : 2);

100         path[0] = asset;
101         path[1] = middleAsset ? uniswapAssetForPath : fundsToken;

103         if(middleAsset) path[2] = fundsToken;

105         uint256[] memory returnAmounts = IUniswapRouter(uniswapRouter).
                swapExactTokensForTokens(
106             assetBalance,
107             minAmount.sub(minAmount.mul(_globals.maxSwapSlippage()).div(10000)),
108             path,
109             address(this),
110             block.timestamp
111         );

113         emit ERC20Conversion(asset, returnAmounts[0], returnAmounts[path.length - 1]);
114     }
```

Listing 3.2: MapleTreasury::convertERC20()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

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

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `78f46ce`.

3.2 Proper Effective Stake Date Calculation

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `StakeLocker`, `PoolLib`
- Category: Business Logic [14]
- CWE subcategory: CWE-841 [10]

Description

The `StakeLocker` contract maintains a timestamp, i.e., the effective stake date, for each active staker. This timestamp value is a weighted representation of the effective single stake date of the user based on their stake amounts. This value is determined using the following equation:

```
coefficient = stakeAmount/(currentStake + stakeAmount)
stakeDate = existingStakeDate + (block.timestamp - existingStakeDate) * coefficient
```

It should be highlighted that the calculation should be performed before the new `stakeAmount` is transferred to be included in `currentStake`. However, this is not followed in current implementation. To elaborate, we show below the implementation of two relevant routines, i.e., `_transfer()` and `_updateStakeDate()`. Note the call to `_updateStakeDate()` (line 216) is made after the new stake amount is transferred to the recipient (line 215).

```
206     /**
207         @dev Transfer StakerLockerFDTs.
208         @param from Address sending   StakeLockerFDTs
209         @param to   Address receiving StakeLockerFDTs
```

```
        @param amt  Amount of FDTs to transfer
    */
    function _transfer(address from, address to, uint256 amt) internal override canUnstake {
        _whenProtocolNotPaused();
        _isAllowed(to);
        super._transfer(from, to, amt);
        _updateStakeDate(to, amt);
    }
```

Listing 3.3: StakeLocker::_transfer()

```
    /**
        @dev Updates information used to calculate unstake delay.
        @param who Staker who deposited BPTs
        @param amt Amount of BPTs staker has deposited
    */
    function _updateStakeDate(address who, uint256 amt) internal {
        uint256 stkDate = stakeDate[who];
        if (stkDate == 0) {
            stakeDate[who] = block.timestamp;
        } else {
            uint256 coef   = WAD.mul(amt).div(balanceOf(who) + amt);
            stakeDate[who] = stkDate.add(((block.timestamp.sub(stkDate)).mul(coef)).div(WAD));  // date + (now - stkDate) * coef
        }
    }
```

Listing 3.4: StakeLocker::_updateStakeDate()

Recommendation Adjust the order inside the affected `_transfer()` routine. An example revision is shown as follows:

```
    /**
        @dev Transfer StakerLockerFDTs.
        @param from Address sending   StakeLockerFDTs
        @param to   Address receiving StakeLockerFDTs
        @param amt  Amount of FDTs to transfer
    */
    function _transfer(address from, address to, uint256 amt) internal override canUnstake {
        _whenProtocolNotPaused();
        _isAllowed(to);
        _updateStakeDate(to, amt);
        super._transfer(from, to, amt);
    }
```

Listing 3.5: StakeLocker::_transfer()

Status The issue has been fixed by this commit: `1a26e9c`.

3.3 Inconsistency Between Document and Implementation

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found in line 211 of `ERC2222::updateFundsReceived()`, line 164 of `ExtendedFDT::updateFundsReceived()`, and line 166 of `BasicFDT::updateFundsReceived()`. Using the `ERC2222::updateFundsReceived()` routine as an example, the preceding function summary indicates that "the contract computes the delta of the previous and the new funds token balance". However, the implemented logic (line 218) indicates it is the delta of the new and the previous funds token balance.

```
209     /**
210      * @dev Register a payment of funds in tokens. May be called directly after a
             deposit is made.
211      * @dev Calls _updateFundsTokenBalance(), whereby the contract computes the delta of
              the previous and the new
212      * funds token balance and increments the total received funds (cumulative) by delta
              by calling _registerFunds()
213      */
214     function updateFundsReceived() public virtual {
215         int256 newFunds = _updateFundsTokenBalance();
216 
217         if (newFunds > 0) {
218             _distributeFunds(newFunds.toUint256Safe());
219         }
220     }
```

Listing 3.6: ERC2222::updateFundsReceived()

Moreover, the function summary of `StakeLocker::canUnstake()` is not accurate. The `canUnstake()` allows for unstaking in the following two conditions: 1) the user is not `Pool Delegate` and the `Pool` is in the `Finalized` state or 2) The `Pool` is in `Initialized` or `Deactivated` state. The current description on the second condition is inaccurate.

```
    /**
        @dev canUnstake enables unstaking in the following conditions:
        1. User is not Pool Delegate and the Pool is in Finalized state.
        2. User is Pool Delegate and the Pool is in Initialized or Deactivated state.
```

```
    */
    modifier canUnstake() {
        require(
            (msg.sender != IPool(owner).poolDelegate() && IPool(owner).poolState() == 1)

            IPool(owner).poolState() == 0  IPool(owner).poolState() == 2,
            "StakeLocker:ERR_STAKE_LOCKED"
        );
        _;
    }
```

Listing 3.7: StakeLocker::canUnstake()

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been fixed by this commit: `45896c2`.

3.4 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [15]
- CWE subcategory: CWE-663 [8]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [22] exploit, and the recent `Uniswap/Lendf.Me` hack [21].

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

Apparently, the interaction with the external contract (line 351) starts before effecting the update on internal states (lines 356−363), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
    function makePayment() external {
        _whenProtocolNotPaused();
        _isValidState(State.Active);
        (uint256 total, uint256 principal, uint256 interest,) = getNextPayment();
        paymentsRemaining--;
        _makePayment(total, principal, interest);
    }

    /**
        @dev Make the full payment for this loan, a.k.a. "calling" the loan. This
            requires the borrower to pay a premium.
    */
    function makeFullPayment() public {
        _whenProtocolNotPaused();
        _isValidState(State.Active);
        (uint256 total, uint256 principal, uint256 interest) = getFullPayment();
        paymentsRemaining = uint256(0);
        _makePayment(total, principal, interest);
    }

    /**
        @dev Internal function to update the payment variables and transfer funds from
            the borrower into the Loan.
    */
    function _makePayment(uint256 total, uint256 principal, uint256 interest) internal {

        _checkValidTransferFrom(loanAsset.transferFrom(msg.sender, address(this), total)
            );

        // Caching it to reduce the `SLOADS`.
        uint256 _paymentsRemaining = paymentsRemaining;
        // Update internal accounting variables.
        if (_paymentsRemaining == uint256(0)) {
            principalOwed   = uint256(0);
            loanState       = State.Matured;
            nextPaymentDue = uint256(0);
        } else {
            principalOwed   = principalOwed.sub(principal);
            nextPaymentDue = nextPaymentDue.add(paymentIntervalSeconds);
        }
        ...
    }
```

Listing 3.8: Loan::makePayment()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`. Note similar issues exist in other contracts, including `Pool::deposit()` and the adherence of `checks-effects-interactions` best practice is recommended in a number of related routines, e.g., `StakingRewards::stake()`, `Loan::unwind()`, `Pool::withdrawFunds()` etc.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been addressed by ensuring the proper vetting process in place so that no `re-entrancy`-capable tokens will be introduced.

3.5 Avoidance Of Zero Amount Transfer

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1041 [1]

Description

A common task in the `Maple` protocol is to manage the asset flow among different component contracts (e.g., `LiquidityLocker`, `FundingLocker`, and `StakeLocker`). For gas optimization purposes, there is no need to make the asset-transferring call if the transferred amount is 0.

To elaborate, we show below the code snippet of `ERC2222::withdrawFunds()`. This routine allows a token holder to withdraw all available funds. However, it also makes the transfer request (line 180) even when the given `withdrawableFunds` is 0.

```
174     /**
175      * @dev Withdraws all available funds for a token holder
176      */
177     function withdrawFunds() public virtual override {
178         uint256 withdrawableFunds = _prepareWithdraw();

180         require(fundsToken.transfer(msg.sender, withdrawableFunds), "FDT:TRANSFER_FAILED
                ");

182         _updateFundsTokenBalance();
183     }
```

Listing 3.9: withdrawFunds()

Note the same issue is also present in other routines, including `ERC2222::withdrawFundsOnBehalf()` and `FDT::withdrawFunds()`.

Recommendation Avoid the token `transfer()` call when the transferred amount is 0.

Status The issue has been fixed by this commit: `04df2cf`.

3.6 Improved Sanity Checks For System/Function Parameters

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MapleGlobals`
- Category: Coding Practices [13]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Maple` protocol is no exception. Specifically, if we examine the `MapleGlobals` contract, it has defined a number of protocol-wide risk parameters, such as `setInvestorFee` and `setTreasuryFee`. In the following, we show the corresponding routines that allow for their changes.

```
    /**
        @dev Adjust investorFee (in basis points). Only Governor can call.
        @param _fee The fee, e.g., 50 = 0.50%
    */
    function setInvestorFee(uint256 _fee) public isGovernor {
        investorFee = _fee;
        emit GlobalsParamSet("INVESTOR_FEE", _fee);
    }

    /**
        @dev Adjust treasuryFee (in basis points). Only Governor can call.
        @param _fee The fee, e.g., 50 = 0.50%
    */
    function setTreasuryFee(uint256 _fee) public isGovernor {
        treasuryFee = _fee;
        emit GlobalsParamSet("TREASURY_FEE", _fee);
    }
```

Listing 3.10: MapleGlobals::setInvestorFee() and MapleGlobals::setTreasuryFee()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `investorFee` may charge unreasonably high fee in the `fundLoan()` operation, hence incurring cost to borrowers or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `b6b4946`.

3.7 Better Handling of Privilege Transfers

- ID: PVE-007
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Targets: MapleGlobals
- Category: Security Features [11]
- CWE subcategory: CWE-282 [4]

Description

Maple implements a rather basic access control mechanism that allows a privileged account, i.e., governor, to be granted exclusive access to typically sensitive functions (e.g., the setting of oracle and fee parameters). Because of the privileged access and the implications of these sensitive functions, the governor account is essential for the protocol-level safety and operation. In the following, we elaborate with the governor account.

Within the governing contract MapleGlobals, a specific function, i.e., setGovernor(), is provided to allow for possible governor updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the _newGovernor parameter is always correctly provided. However, in the unlikely situation, when an incorrect _newGovernor is provided, the contract owner may be forever lost, which might be devastating for Maple operation and maintenance.

As a common best practice, instead of achieving the governor update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the governor update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract governor to an uncontrolled address. In other words, this two-step procedure ensures that a governor public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the governor transfer process.

```
    /**
        @dev Set a new Governor. Only Governor can call.
        @param _newGovernor Address of new Governor
    */
    function setGovernor(address _newGovernor) public isGovernor {
        require(_newGovernor != address(0), "MapleGlobals:ZERO_ADDRESS_GOVERNOR");
        governor = _newGovernor;
        emit GlobalsAddressSet("GOVERNOR", _newGovernor);
    }
```

Listing 3.11: MapleGlobals::setGovernor()

Recommendation Implement a two-step approach for governor update (or transfer): `setGovernor()` and `acceptGovernor()`.

Status The issue has been fixed by this commit: `963ca89`.

3.8 Possible Front-Running For Reduced Stake Requirements

- ID: PVE-008
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Pool`
- Category: Time and State [16]
- CWE subcategory: CWE-682 [9]

Description

`Pool Delegates` are in charge of managing the `Pool`'s liquidity. In order to open a `Pool`, the `Pool Delegate` is required to be whitelisted by `MapleDAO`. After that, the `Pool Delegate` must stake at least the minimum amount of BPTs required to meet the level of `Pool` coverage specified by `MapleDAO`. Once the `Pool` has been finalized, the `Pool Delegate` can start earning a portion of the interest earned using the pool capital as well as the `investorFee` (Section 3.6).

In the following, we show the `Pool::finalize()` routine, which can only be invoked by the approved `Pool Delegate` to open the `Pool`. This routine ensures that the `Pool Delegate` has the required minimum amount of BPTs staked in `StakeLocker`.

```
    function finalize() external {
        _whenProtocolNotPaused();
        _isValidState(State.Initialized);
        _isValidDelegate();
        (,, bool stakePresent,,) = getInitialStakeRequirements();
        require(stakePresent, "Pool:INSUFFICIENT_STAKE");
        poolState = State.Finalized;
        emit PoolStateChanged(poolState);
    }
```

Listing 3.12: Pool:: finalize ()

However, it comes to our attention that the pool share requirement is computed by calling `bPool.calcPoolInGivenSingleOut()` (line 170). As the trading pool may be manipulated and an imbalanced pool can be crafted to lead to a much smaller staking requirement from the (manipulated) assessment.

```
153    function getPoolSharesRequired(
154        address _bPool,
155        address liquidityAsset,
156        address staker,
157        address stakeLocker,
```

```
        uint256 liquidityAssetAmountRequired
    ) public view returns (uint256, uint256) {

        IBPool bPool = IBPool(_bPool);

        uint256 tokenBalanceOut = bPool.getBalance(liquidityAsset);
        uint256 tokenWeightOut  = bPool.getDenormalizedWeight(liquidityAsset);
        uint256 poolSupply      = bPool.totalSupply();
        uint256 totalWeight     = bPool.getTotalDenormalizedWeight();
        uint256 swapFee         = bPool.getSwapFee();

        // Fetch amount of BPTs required to burn to receive liquidityAssetAmountRequired
        uint256 poolAmountInRequired = bPool.calcPoolInGivenSingleOut(
            tokenBalanceOut,
            tokenWeightOut,
            poolSupply,
            totalWeight,
            liquidityAssetAmountRequired,
            swapFee
        );

        // Fetch amount staked in stakeLocker by staker
        uint256 stakerBalance = IERC20(stakeLocker).balanceOf(staker);

        return (poolAmountInRequired, stakerBalance);
    }
```

Listing 3.13: PoolLib::getPoolSharesRequired()

It is important to emphasize this issue may occur in other contexts. Specifically, in a similar sandwich-based attack against `handleDefault()/exitswapExternAmountOut()`, the protocol may gain less or loss more due to manipulated trade price. For example, a malicious sandwich attack may foil the above validation with minimum amount of BPTs to prevent the pool from being finalized. Also, a number of other routines, i.e., `BPTVal()/getSwapOutValue()/getSwapOutValueLocker()/getPoolSharesRequired()`, are similarly affected due to the external DEX interaction.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider measuring the stability of involved pools or relying on a trustworthy oracle. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to ensure `Pool Delegate` is sufficiently staked before opening up a `Pool`.

Status The issue has been fixed by this commit: `ee820b1`. The team also clarifies that the `Pool Delegate` entity is trusted in current protocol design.

3.9 Improved Precision By Multiplication And Division Reordering

- ID: PVE-009
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [17]
- CWE subcategory: CWE-190 [3]

Description

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

In particular, we use the `DebtLocker::calcAllotment()` as an example. This routine is used to calculate the resulting allotment of a particular claim.

```
38    function calcAllotment(uint256 newAmt, uint256 totalNewAmt, uint256 totalClaim)
          internal pure returns (uint256) {
39        return newAmt.mul(WAD).div(totalNewAmt).mul(totalClaim).div(WAD);
40    }
```

Listing 3.14: DebtLocker::calcAllotment()

We notice the calculation of the resulting allotment (line 39) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `newAmt.mul(totalClaim).div(totalNewAmt).mul(totalClaim)`. Similarly, the calculation of `calcMinAmount()` in `Util` contract (lines 26 − 29) can be accordingly adjusted. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible. Note the `Util::calcMinAmount()` routine can be similarly improved.

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

Status The issue has been fixed by this commit: `c24515a`.

3.10 Simplification of PoolLib::updateDepositDate()

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PoolLib, StakeLocker
- Category: Numeric Errors [17]
- CWE subcategory: CWE-190 [3]

Description

As mentioned in Section 3.9, SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce the subtle, but troublesome issue of precision loss.

In the following, we show the PoolLib::updateDepositDate() routine. This routine is designed to compute and update the effective deposit date for the depositing user. We notice that the current implementation introduces the scaling factor WAD to compute an internal coefficient. However, by better re-arrangement of the calculation order, we can avoid the use the scaling factor without any precision loss.

```
/**
    @dev Update the effective deposit date based on how much new capital has been
        added.
         If more capital is added, the depositDate moves closer to the current
            timestamp.
    @param  depositDate Weighted timestamp representing effective deposit date
    @param  balance     Balance of PoolFDT tokens of user
    @param  amt         Total deposit amount
    @param  who         Address of user depositing
*/
function updateDepositDate(mapping(address => uint256) storage depositDate, uint256
    balance, uint256 amt, address who) internal {
    if (depositDate[who] == 0) {
        depositDate[who] = block.timestamp;
    } else {
        uint256 depDate  = depositDate[who];
        uint256 coef     = (WAD.mul(amt)).div(balance + amt);
        depositDate[who] = (depDate.mul(WAD).add((block.timestamp.sub(depDate)).mul(
            coef))).div(WAD);  // depDate + (now - depDate) * coef
    }
}
```

Listing 3.15: PoolLib::updateDepositDate()

Specifically, the deposit date can be computed as follows:

```
/**
```

```
        @dev Update the effective deposit date based on how much new capital has been added.
             If more capital is added, the depositDate moves closer to the current timestamp.
        @param  depositDate Weighted timestamp representing effective deposit date
        @param  balance     Balance of PoolFDT tokens of user
        @param  amt         Total deposit amount
        @param  who         Address of user depositing
    */
    function updateDepositDate(mapping(address => uint256) storage depositDate, uint256 balance, uint256 amt, address who) internal {
        if (depositDate[who] == 0) {
            depositDate[who] = block.timestamp;
        } else {
            uint256 depDate  = depositDate[who];
            uint256 dTime    = block.timestamp.sub(depDate);
            depositDate[who] = depDate.add(dTime.mul(amt).div(balance + amt));  // depDate + (now - depDate) * (amt / (balance + amt))
        }
    }
```

Listing 3.16: PoolLib::updateDepositDate()

Note a similar optimization is also present in `StakeLocker::_updateStakeDate()`.

Recommendation Revise the aforementioned routines with an optimized version.

Status The issue has been fixed by this commit: `021912f`.

3.11 Timely updateFundsReceived() in Loan Management

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Loan`
- Category: Business Logic [14]
- CWE subcategory: CWE-841 [10]

Description

The `Loan` contract provides a number of core routines for supplying/borrowing users to interact with, including `unwind()`, `drawdown()`, `triggerDefault()`, `makePayment()`, `makeFullPayment()`, and etc. To facilitate the execution of each core routine, Maple adopts the `FundsDistributionToken` (`FDT`) as the key for proper accounting and attribution. Note that `FDT` is compliant with ERC20 and additionally implements the ERC2222 token standard.

During the entire lifecycle of a `loan`, there are five different states: `Live`, `Active`, `Matured`, `Expired`, and `Liquidated`. In the following, we examine the `unwind()` routine that transitions the `loan` from `Live` to Expired.

```
    /**
        @dev If the borrower has not drawn down on the Loan past the drawdown grace
            period, return capital to Loan,
             where it can be claimed back by LoanFDT holders.
    */
    function unwind() external {
        _whenProtocolNotPaused();
        _isValidState(State.Live);

        // Update accounting for claim(), transfer funds from FundingLocker to Loan
        excessReturned += LoanLib.unwind(loanAsset, superFactory, fundingLocker,
            createdAt);

        // Transition state to Expired
        loanState = State.Expired;
    }
```

Listing 3.17: Loan::unwind()

The unwind() implements a rather straightforward logic by returning the capital to the loan contract if the borrower has not drawn down on the loan past the drawdown grace period. However, it also comes to our attention it does not timely updating the LoanFDT accounting with received capital back to the loan. As a result, if a lender attempts to withdrawFunds(), the funds accounted for may not reflect the just-returned capital.

Note the maple-token repository contains the MPL protocol token implementation that shares the similar issue in the ERC2222::withdrawFunds() /ERC2222::withdrawFundsOnBehalf() routines.

Recommendation Timely invoke updateFundsReceived() whenever any fund is returned back to loan from possible unwind(), drawdown(), and makePayment()/makeFullPayment(), or triggerDefault (). An example revision to the above code snippet is shown below.

```
    /**
        @dev If the borrower has not drawn down on the Loan past the drawdown grace
            period, return capital to Loan,
             where it can be claimed back by LoanFDT holders.
    */
    function unwind() external {
        _whenProtocolNotPaused();
        _isValidState(State.Live);

        // Update accounting for claim(), transfer funds from FundingLocker to Loan
        excessReturned += LoanLib.unwind(loanAsset, superFactory, fundingLocker,
            createdAt);

        updateFundsReceived();

        // Transition state to Expired
        loanState = State.Expired;
```

```
    }
```

Listing 3.18: Loan::unwind()

Status The issue has been fixed by this commit: 84a1989.

3.12 Lack Of Proper Enforcement Of fundingPeriodSeconds

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Loan
- Category: Business Logic [14]
- CWE subcategory: CWE-841 [10]

Description

As mentioned in Section 3.11, during the entire lifecycle of a `loan`, there are five different states: `Live`, `Active`, `Matured`, `Expired`, and `Liquidated`. We have so far examined the `unwind()` routine that transitions the `loan` from `Live` to `Expired`. In the following, we examine another routine `drawdown()` that transitions the `loan` from `Live` to `Active`.

To elaborate, we show below the `drawdown()` implementation. It comes to our attention that once a `loan` is funded (i.e., in `Live` state), there is a time period called `fundingPeriodSeconds` that allows the intended borrower to draw down on their `loan`. Our analysis shows that current implementation does not enforce the logic in disallowing the `loan` drawdown after the `fundingPeriodSeconds` period expires. Note that the `fundingPeriodSeconds` is specified when the `loan` contract is instantiated.

```
    function drawdown(uint256 amt) external {
        _whenProtocolNotPaused();
        _isValidBorrower();
        _isValidState(State.Live);
        IGlobals globals = _globals(superFactory);

        IFundingLocker _fundingLocker = IFundingLocker(fundingLocker);

        require(amt >= requestAmount,                "Loan:AMT_LT_MIN_RAISE");
        require(amt <= _getFundingLockerBalance(),   "Loan:AMT_GT_FUNDED_AMT");

        // Update the principal owed and drawdown amount for this loan.
        principalOwed  = amt;
        drawdownAmount = amt;

        loanState = State.Active;

        // Transfer the required amount of collateral for drawdown from Borrower to
            CollateralLocker.
```

```
        _checkValidTransferFrom(collateralAsset.transferFrom(borrower, collateralLocker,
             collateralRequiredForDrawdown(amt)));

        // Transfer funding amount from FundingLocker to Borrower, then drain remaining
            funds to Loan.
        uint256 treasuryFee = globals.treasuryFee();
        uint256 investorFee = globals.investorFee();

        address treasury = globals.mapleTreasury();

        feePaid             = amt.mul(investorFee).div(10000);  // Update fees paid for
            claim()
        uint256 treasuryAmt = amt.mul(treasuryFee).div(10000);  // Calculate amt to send
             to MapleTreasury

        _transferFunds(_fundingLocker, treasury,      treasuryAmt);
                                   // Send treasuryFee directly to MapleTreasury
        _transferFunds(_fundingLocker, address(this), feePaid);
                                        // Transfer `feePaid` to the this i.e Loan
            contract
        _transferFunds(_fundingLocker, borrower,      amt.sub(treasuryAmt).sub(feePaid))
            ;  // Transfer drawdown amount to Borrower

        // Update excessReturned for claim()
        excessReturned = _getFundingLockerBalance();

        // Drain remaining funds from FundingLocker (amount equal to excessReturned)
        require(_fundingLocker.drain(), "Loan:DRAIN");

        _emitBalanceUpdateEventForCollateralLocker();
        _emitBalanceUpdateEventForFundingLocker();
        _emitBalanceUpdateEventForLoan();

        emit BalanceUpdated(treasury, address(loanAsset), loanAsset.balanceOf(treasury))
            ;

        emit Drawdown(amt);
    }
```

Listing 3.19: Loan::drawdown()

Recommendation Honor the `fundingPeriodSeconds` parameter that is specified when the `loan` contract is instantiated and ensure the `loan` drawdown occurs within the given time period.

Status The issue has been resolved and the team clarifies that it is purposely not enforced in the `drawdown()` function in the case that the funders wanted to give the borrower some extra time to draw down.

3.13 Redundant Code Removal

- ID: PVE-013
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Pool`
- Category: Coding Practices [13]
- CWE subcategory: CWE-563 [7]

Description

Maple makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Pausable`, to facilitate its code implementation and organization. For example, the `Pool` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `Pool::constructor()` implementation, there is an internal helper routine for the instantiation of a new `StakeLocker` (line 125). This routine can be optimized away.

```
85      constructor(
86          address  _poolDelegate,
87          address  _liquidityAsset,
88          address  _stakeAsset,
89          address  _slFactory,
90          address  _llFactory,
91          uint256  _stakingFee,
92          uint256  _delegateFee,
93          uint256  _liquidityCap,
94          string memory name,
95          string memory symbol
96      ) PoolFDT(name, symbol) public {
97          require(_globals(msg.sender).isValidLoanAsset(_liquidityAsset), "Pool:INVALID_LIQ_ASSET");
98          require(_liquidityCap != uint256(0),                             "Pool:INVALID_CAP");
99
100         // NOTE: Max length of this array would be 8, as thats the limit of assets in a balancer pool
101         address[] memory tokens = IBPool(_stakeAsset).getFinalTokens();
102
103         uint256  i = 0;
104         bool valid = false;
105
106         // Check that one of the assets in balancer pool is liquidityAsset
107         while(i < tokens.length && !valid) { valid = tokens[i] == _liquidityAsset; i++; }
108
109         require(valid, "Pool:INVALID_STAKING_POOL");
```

```
110
111         // Assign variables relating to liquidityAsset
112         liquidityAsset         = IERC20(_liquidityAsset);
113         liquidityAssetDecimals = ERC20(_liquidityAsset).decimals();
114
115         // Assign state variables
116         stakeAsset    = _stakeAsset;
117         slFactory     = _slFactory;
118         poolDelegate = _poolDelegate;
119         stakingFee    = _stakingFee;
120         delegateFee   = _delegateFee;
121         superFactory = msg.sender;
122         liquidityCap = _liquidityCap;
123
124         // Initialize the LiquidityLocker and StakeLocker
125         stakeLocker     = createStakeLocker(_stakeAsset, _slFactory, _liquidityAsset,
                _globals(msg.sender));
126         liquidityLocker = address(ILiquidityLockerFactory(_llFactory).newLocker(
                _liquidityAsset));
127
128         // Withdrawal penalty default settings
129         principalPenalty = 500;
130         penaltyDelay     = 30 days;
131         lockupPeriod     = 90 days;
132
133         emit PoolStateChanged(poolState);
134     }
135
136     /**
137         @dev Deploys and assigns a StakeLocker for this Pool (only used once in
                constructor).
138         @param _stakeAsset     Address of the asset used for staking
139         @param _slFactory      Address of the StakeLocker factory used for instantiation
140         @param _liquidityAsset Address of the liquidity asset, required when burning
                _stakeAsset
141         @param globals         IGlobals for Maple Globals contract
142     */
143     function createStakeLocker(address _stakeAsset, address _slFactory, address
            _liquidityAsset, IGlobals globals) private returns (address) {
144         require(IBPool(_stakeAsset).isBound(globals.mpl()) && IBPool(_stakeAsset).
                isFinalized(), "Pool:INVALID_BALANCER_POOL");
145         return IStakeLockerFactory(_slFactory).newLocker(_stakeAsset, _liquidityAsset);
146     }
```

Listing 3.20: Pool::constructor()

In particular, the validation of `IBPool(_stakeAsset).getFinalTokens()` (line 101) has already guaranteed the `BPool` is finalized. In other words, the validation on `IBPool(_stakeAsset).isFinalized()` (line 144) becomes redundant and thus can be safely removed.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `e80e0ef`.

3.14 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [14]
- CWE subcategory: CWE-841 [10]

Description

In Maple, the `Pool` contract is designed to be the main entry for interaction with supplying users. In particular, one entry routine, i.e., `deposit()`, accepts asset transfer-in and mints the corresponding `LP` tokens to represent the depositor's share in the lending pool. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of Maple. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    /**
        @dev Liquidity providers can deposit liquidityAsset into the LiquidityLocker,
            minting FDTs.
        @param amt Amount of liquidityAsset to deposit
    */
    function deposit(uint256 amt) external {
        _whenProtocolNotPaused();
        _isValidState(State.Finalized);
        require(isDepositAllowed(amt), "Pool:LIQUIDITY_CAP_HIT");
        require(liquidityAsset.transferFrom(msg.sender, liquidityLocker, amt), "Pool:
            DEPOSIT_TRANSFER_FROM");
        uint256 wad = _toWad(amt);

        PoolLib.updateDepositDate(depositDate, balanceOf(msg.sender), wad, msg.sender);
        _mint(msg.sender, wad);
        _emitBalanceUpdatedEvent();
    }
```

Listing 3.21: Pool::deposit()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Maple for borrowing/lending. In fact, Maple is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been acknowledged by the team. And the team has a proper vetting process in place to prevent deflationary/rebasing tokens from being listed in the protocol.

3.15 Bypass of lockupPeriod in Pool::withdraw()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Pool`
- Category: Business Logic [14]
- CWE subcategory: CWE-841 [10]

Description

By design, the Maple protocol will generate and collect fees that are attributed to liquidity providers (LPs). Also, due to the fact that the interest earned by the Maple protocol is accrued in discrete large payments of interest rather that steady streams of income, it is important to prevent any possibility for malicious LPs to exploit the interest distribution mechanism in Pools.

With that, the Maple protocol requires LPs to go through a `Pool Lockup Period` that specifies if a user has not waited a specified period of time after deposit, the user cannot withdraw. Note the current protocol specifies 90 days as the `Pool Lockup Period`. For each LP account, the associated lockup period is recorded as `[depositDate[account], depositDate[account].add(lockupPeriod)]`.

To elaborate, we show below the `Pool::withdraw()` routine that properly enforces the requirement of `require(depositDate[msg.sender].add(lockupPeriod)<= block.timestamp)` (line 238). However, it comes to our attention that the `poolFDT` token lacks the implementation to properly keep track of the `depositDate` state when the token is being transferred! As a result, a LP can completely avoid the lockup period by simply transferring the assets to another new account and withdrawing the transferred funds from the new account without penalty!

```
233     function withdraw(uint256 amt) external {
234         _whenProtocolNotPaused();
235         uint256 wad    = _toWad(amt);
236         uint256 fdtAmt = totalSupply() == wad && amt > 0 ? wad - 1 : wad;  // If last withdraw, subtract 1 wei to maintain FDT accounting
237         require(balanceOf(msg.sender) >= fdtAmt, "Pool:USER_BAL_LT_AMT");
238         require(depositDate[msg.sender].add(lockupPeriod) <= block.timestamp, "Pool:FUNDS_LOCKED");
239
240         uint256 allocatedInterest = withdrawableFundsOf(msg.sender);                                     // FDT accounting interest
241         uint256 recognizedLosses  = recognizableLossesOf(msg.sender);                                    // FDT accounting losses
242         uint256 priPenalty        = principalPenalty.mul(amt).div(10000);                                // Calculate flat principal penalty
243         uint256 totPenalty        = calcWithdrawPenalty(allocatedInterest.add(priPenalty), msg.sender);  // Calculate total penalty
244
245         // Amount that is due after penalties and realized losses are accounted for.
246         // Total penalty is distributed to other LPs as interest, recognizedLosses are absorbed by the LP.
247         uint256 due = amt.sub(totPenalty).sub(recognizedLosses);
248
249         _burn(msg.sender, fdtAmt);  // Burn the corresponding FDT balance
250         recognizeLosses();          // Update loss accounting for LP,  decrement `bptShortfall`
251         withdrawFunds();            // Transfer full entitled interest, decrement `interestSum`
252
253         interestSum = interestSum.add(totPenalty);  // Update the `interestSum` with the penalty amount
254         updateFundsReceived();                      // Update the `pointsPerShare` using this as fundsTokenBalance is incremented by `totPenalty`
255
256         // Transfer amt - totPenalty - recognizedLosses
257         require(ILiquidityLocker(liquidityLocker).transfer(msg.sender, due), "Pool::WITHDRAW_TRANSFER");
258
259         _emitBalanceUpdatedEvent();
260     }
```

Listing 3.22: Pool::withdraw()

Recommendation Properly record the `depositDate` when a `LP` holder transfers the `poolFDT` token.

Status The issue has been fixed by this commit: `b0b4815`.

3.16 Suggested Addition of rescueToken()

- ID: PVE-016
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [14]
- CWE subcategory: CWE-841 [10]

Description

By design, the Maple protocol has developed a number of lockers that hold various types of assets. From past experience with current popular DeFi protocols, e.g., `YFI/Curve`, we notice that there is always non-trivial possibilities that non-related tokens may be accidentally sent to the pool contract(s). To avoid unnecessary loss of `Maple` users, we suggest to add necessary support of rescuing tokens accidentally sent to the contract. This is a design choice for the benefit of protocol users.

Recommendation Add the support of rescuing tokens accidentally sent to the contract.

Status This issue has been resolved and the team has implemented a `rescueToken()` solution for `Pool` and `Loan` in the following commit: `e01199c`.

3.17 Potential Collusion Between PoolDelegate And Borrowers

- ID: PVE-017
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `LendingPoolConfigurator`
- Category: Time and State [12]
- CWE subcategory: CWE-362 [6]

Description

The current protocol is designed with an implicit trust on the approved `Pool Delegates`. However, there is still a need to properly verify the operations to protect user funds. Specifically, as mentioned in Section 3.8, `Pool Delegates` are in charge of managing the `Pool`'s liquidity. In order to open a `Pool`, the `Pool Delegate` is required to be whitelisted by `MapleDAO`. After that, the `Pool Delegate` must stake at least the minimum amount of BPTs required to meet the level of `Pool` coverage specified by MapleDAO. Once the Pool has been finalized, the Pool Delegate can start earning a portion of the interest earned using the pool capital as well as the investorFee.

The Borrowers can request capital from the platform by instantiating a Loan contract with the intended loan terms. As the manager of the pool, the Pool Delegate is supposed to perform due diligence and agree terms with Borrowers. Once these loan terms are agreed between the Borrower and the Pool Delegate, the Borrower can withdraw the requested funds for a fixed term, at a fixed rate, and at a fixed collateralization level.

However, it brings up a possible collusion situation where the Borrower is an accomplice. In other words, the Borrower can simply create a loan that attempts to request all funds available in the pool and the loan request can then be funded by the Pool Delegate. Note in this collusion situation, by current protocol design, the only stake for the Pool Delegate is the amount of BPTs required to meet the level of Pool coverage (specified by MapleDAO). Moreover, though the total liquidity in the pool may be limited by liquidityCap, this liquidityCap parameter can be adjusted by the Pool Delegate.

Recommendation Revise the current protocol design to defend against the above collusion situation.

Status This issue has been resolved. As mentioned in Section 3.8, current protocol, by design, considers Pool Delegates are trusted actors.

3.18 Revisited Assumption on Trusted Governance

- ID: PVE-018
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: MapleGolbals
- Category: Security Features [11]
- CWE subcategory: CWE-287 [5]

Description

In the Maple protocol, the governance account plays a critical role in governing and regulating the system-wide operations (e.g., factory contract whitelisting, oracle addition, fee adjustment, and parameter setting). It also has the privilege to regulate or govern the flow of assets for borrowing and lending among the involved components, i.e., LiquidityLocker, FundlingLocker, and StakeLocker.

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

```
    /**
        @dev Update the valid PoolFactory mapping. Only Governor can call.
        @param poolFactory Address of PoolFactory
        @param valid       The new bool value for validating poolFactory
```

```
    */
    function setValidPoolFactory(address poolFactory, bool valid) external isGovernor {
        isValidPoolFactory[poolFactory] = valid;
    }

    /**
        @dev Update the valid PoolFactory mapping. Only Governor can call.
        @param loanFactory Address of LoanFactory
        @param valid       The new bool value for validating loanFactory.
    */
    function setValidLoanFactory(address loanFactory, bool valid) external isGovernor {
        isValidLoanFactory[loanFactory] = valid;
    }

    /**
        @dev Set the validity of a subFactory as it relates to a superFactory. Only
            Governor can call.
        @param superFactory The core factory (e.g. PoolFactory, LoanFactory)
        @param subFactory   The sub factory used by core factory (e.g.
            LiquidityLockerFactory)
        @param valid        The validity of subFactory within context of superFactory
    */
    function setValidSubFactory(address superFactory, address subFactory, bool valid)
        external isGovernor {
        validSubFactories[superFactory][subFactory] = valid;
    }
```

Listing 3.23: Various Setters in MapleGlobals

We emphasize that the privilege assignment with various factory contracts is necessary and required for proper protocol operations. However, it is worrisome if the `governance` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised `governance` account would allow the attacker to add a malicious `calculator` or change other settings to steal funds in current protocol, which directly undermines the assumption of the `Maple` protocol.

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

Status This issue has been confirmed and partially mitigated with a multi-sig account to regulate the governance privileges.

4 Conclusion

In this audit, we have analyzed the Maple design and implementation. The system presents a unique, robust offering as a decentralized non-custodial corporate credit market that aims to provide capital to institutional borrowers through globally accessible fixed-income yield opportunities. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[4] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

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

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

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

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

[9] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

[12] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

[16] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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

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