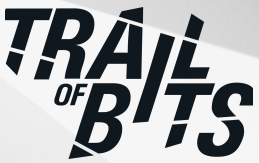

Maple Finance v2

Security Assessment

December 14, 2022

Prepared for:
Lucas Manuel
Maple Labs

Prepared by: **Simone Monica and Robert Schneider**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Maple Labs engaged Trail of Bits to review the security of its smart contracts. From August 29 to September 30, 2022, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Additionally, from November 21 to November 23, 2022, a consultant conducted an additional security review of updates to the client-provided source code, with three person-days of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system. We had access to the source code and documentation. We performed static and manual testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
Medium	2
Low	9
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	2
Data Validation	6
Denial of Service	1
Undefined Behavior	3

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-MPL-9: Attackers can prevent the pool manager from finishing liquidation**
  An attacker can ensure a liquidation never completes by sending the minimal amount of the collateral token to the liquidator address.

- **TOB-MPL-8: Inaccurate accounting of unrealizedLosses during default warning revert**
  An accounting discrepancy fails to decrement `netLateInterest` from `unrealizedLosses`, which causes `unrealizedLosses` to accrue an over-inflated value.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Simone Monica, Consultant
simone.monica@trailofbits.com

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 24, 2022	Pre-project kickoff call
September 1, 2022	Status update meeting #1
September 9, 2022	Status update meeting #2
September 23, 2022	Status update meeting #3
September 30, 2022	Delivery of report draft
September 30, 2022	Report readout meeting
October 20, 2022	Delivery of final report
December 12, 2022	Delivery of updated final report
December 14, 2022	Delivery of fix report

Project Goals

The engagement was scoped to provide a security assessment of the Maple Labs's protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Could an attacker steal funds from the system?
- Are there appropriate access controls in place?
- Are user-provided parameters sufficiently validated?
- Can a borrower avoid being liquidated?
- Are there front-running or denial-of-service (DoS) opportunities in the system?
- Are the migration code and process safe from exploitation?
- Did updates and fixes introduce any new bugs to the system?

Project Targets

The engagement involved a review and testing of the targets listed below. During the engagement multiple versions were delivered.

Globals-v2

Repository	https://github.com/maple-labs/globals-v2
Initial version	`a094c837d3884ed4f12ad6be31aaec358180bc63`
Final version	`5c65a69f2f461c0a6b5b7960cad71b95047b6597`
Type	Solidity
Platform	Ethereum

Loan

Repository	https://github.com/maple-labs/loan
Initial version	`36fcff2abc00d188c1132d06d4634cbe655a2b69`
Updated version	`33bc494f6db477795e439c7af484d602ffbdeabf`
Final version	`e76c817978f80760f3e1cd8900011a7d7e10137c`
Type	Solidity
Platform	Ethereum

Pool-v2

Repository	https://github.com/maple-labs/pool-v2
Initial version	`17a321f0af8fa2e5bf8ca5b6199fd2b8641bc26c`
Updated version	`ff1ec915a329da9418e9ca36369d089228ad4549`
Final version	`0b52c513076894d728f108b29e5e5cfbd59a2df8`
Type	Solidity
Platform	Ethereum

Withdrawal-manager

Repository	https://github.com/maple-labs/withdrawal-manager
Initial version	`fee2425254cd789f7c108500bd4720fdb11b6c90`
Updated version	`e656fa459291613f3469ab40e381f36deb4eb3f8`

Final version	`f8cfc80a9cb23053ec5f9b127cb55795e6df6d0a`
Type	Solidity
Platform	Ethereum

Liquidations

Repository	https://github.com/maple-labs/liquidations
Initial version	`773f87b4836dd48ba64d58270760538723a9a972`
Updated version	`eb97933eb038d508bbcfa9055ac58bd9047b4782`
Final version	`56da3118036639630d41b48547f2604f9bb61ee6`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches include the following:

MapleGlobals. This contract holds global variables for the system. We looked for incorrect access controls and correct data validation, focusing on setter functions and the price retrieved by the oracle.

MapleLoan. This contract is used by the borrower, who is a trusted actor in the system, to manage the loan (e.g., by adding collateral, obtaining funds, or proposing new terms). We looked for incorrect access controls that would allow a third-party actor to execute actions on the borrower's behalf and for correct accounting of the loan state.

LoanManager. This contract implements the logic for all loan accounting. We investigated potentially incorrect system states, such as those that may occur when advancing the payment accounting or when updating the issuance parameters.

Pool. This pool contract implements EIP-4626. We looked for the correct rounding of operations and for possible non-compliant behavior with EIP-4626.

PoolManager. This contract allows the pool delegate to manage the associated pool and implements functions to process withdrawals for the pool. We looked for incorrect access controls and the correct validation of input data.

TransitionLoanManager. This contract is used to migrate active loans from the old to the new protocol version. We reviewed the contract for the correct addition of loan amounts, such as current loan payments, to the new contract.

WithdrawalManager. This contract contains the logic for liquidity providers to withdraw their funds. We reviewed it for possible exit misconfigurations and for ways by which the users could avoid the locking mechanism.

Liquidator. This helper contract makes liquidations. We looked for ways by which an actor could steal collateral from the liquidation process or avoid a successful liquidation.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

Upgradeability mechanism. We reviewed the upgradeability mechanism only to understand it; we did not look for security flaws.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The system performs mostly simple arithmetic operations. However, its complex underlying bookkeeping—in particular, the code associated with loan accounting—would benefit from further documentation and testing.	**Satisfactory**
Auditing	Certain critical administrative functionalities do not emit events (TOB-MPL-3) or emit them with incorrect values (TOB-MPL-4). The Maple team indicated they use Tenderly as a monitoring plan, but it is unclear if an incident response plan exists.	**Moderate**
Authentication / Access Controls	There appear to be appropriate access controls set for critical state-changing functions. The governor role has global privilege on the system, and a pool delegate has important privileges for its pool. However, there is no clear documentation of what the roles should be able to do.	**Satisfactory**
Complexity Management	The codebase is complex. The logic is split into different smart contracts, which helped the review. However, the codebase would still benefit from more extensive documentation of the architecture (e.g., the expected call flows and the entry-points).	**Satisfactory**
Decentralization	The governor role is a multisig that can control global parameters but does not have power to steal users' funds. The pool delegate is a trusted actor that manages a specific pool; for example, it can decide which loan to fund and can accept new terms for a loan. However, risks	**Moderate**

	associated with privileged actors should be outlined in detail in the documentation. Additionally, users can exit the system as long as the protocol is not paused.	
Documentation	We were provided with only a high-level diagram of the system, along with some supplemental resources. However, detailed technical documentation is still needed. The NatSpec comments and the inline code comments do not always adequately describe the system's behavior.	**Moderate**
Front-Running Resistance	The system's architecture does not allow possible front-running opportunities between the borrower and lender. However, we found an issue during the liquidation process.	**Satisfactory**
Low-Level Manipulation	Assembly is used in only one instance and is justified. The use of low-level calls is limited and includes contract existence checks.	**Strong**
Testing and Verification	Although the codebase does have unit and integration tests and uses a static analyzer tool, we encountered a number of issues that could have been found by a deeper test suite.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Incorrect argument passed to _getPlatformOriginationFee	Undefined Behavior	Low
2	The protocol could stop working prematurely	Data Validation	Low
3	Insufficient event generation	Auditing and Logging	Low
4	Incorrect GovernorshipAccepted event argument	Auditing and Logging	Low
5	Partially incorrect Chainlink price feed safety checks	Data Validation	Low
6	Incorrect implementation of EIP-4626	Undefined Behavior	Low
7	setAllowedSlippage and setMinRatio functions are unreachable	Undefined Behavior	Low
8	Inaccurate accounting of unrealizedLosses during default warning revert	Data Validation	Medium
9	Attackers can prevent the pool manager from finishing liquidation	Denial of Service	Medium
10	WithdrawalManager can have an invalid exit configuration	Data Validation	Low
11	Loan can be impaired when the protocol is paused	Data Validation	Undetermined

12	Fee treasury could go to the zero address	Data Validation	**Low**

Detailed Findings

1. Incorrect argument passed to _getPlatformOriginationFee

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MPL-1
Target: `loan/contracts/MapleLoanFeeManager.sol`	

Description

The `getOriginationFees` function incorrectly uses `msg.sender` instead of the `loan_` parameter. As a result, it returns an incorrect result to users who want to know how much a loan is paying in origination fees.

```
   function getOriginationFees(address loan_, uint256 principalRequested_) external view
override returns (uint256 originationFees_) {
        originationFees_ = _getPlatformOriginationFee(msg.sender, principalRequested_) +
delegateOriginationFee[msg.sender];
    }
```

Figure 1.1: `getOriginationFees` function (`loan/contracts/MapleLoanFeeManager.sol#147-149`)

Exploit Scenario

Bob, a borrower, wants to see how much his loan is paying in origination fees. He calls `getOriginationFees` but receives an incorrect result that does not correspond to what the loan actually pays.

Recommendations

Short term, correct the `getOriginationFees` function to use `loan_` instead of `msg.sender`.

Long term, add tests for view functions that are not used inside the protocol but are intended for the end-users.

2. The protocol could stop working prematurely	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-MPL-2
Target: `pool-v2/contracts/LoanManager.sol-TransitionLoanManager.sol`	

Description

The `_uint48` function is incorrectly implemented such that it requires the input to be less than or equal to `type(uint32).max` instead of `type(uint48).max`. This could lead to the incorrect reversion of successful executions.

```
function _uint48(uint256 input_) internal pure returns (uint32 output_) {
     require(input_ <= type(uint32).max, "LM:UINT32_CAST_OOB");
     output_ = uint32(input_);
 }
```

Figure 2.1: `_uint48` function (`pool-v2/contracts/LoanManager.sol#774-777`)

The function is mainly used to keep track of when each loan's payment starts and when it is due. All variables for which the result of `_uint48` is assigned are effectively of `uint48` type.

Exploit Scenario

The protocol stops working when we reach a `block.timestamp` value of `type(uint32).max` instead of the expected behavior to work until the `block.timestamp` reaches a value of `type(uint48).max`.

Recommendations

Short term, correct the `_uint48` implementation by checking that the `input_` is less than the `type(uint48).max` and that it returns an `uint48` type.

Long term, improve the unit-tests to account for extreme but valid states that the protocol supports.

3. Insufficient event generation	
Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-MPL-3
Target: `globals-v2/contracts/MapleGlobals.sol`	

Description
Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions. This may lead to missed discovery of malfunctioning contracts or malicious attacks.

Multiple critical operations do not emit events. As a result, it will be difficult to review the correct behavior of the contracts once they have been deployed.

The following operations should trigger events:

- Constructor should emit `DefaultTimelockParametersSet` event
- `setPriceOracle`
- `setManualOverridePrice`

Exploit Scenario
The Maple team must use the `setManualOverridePrice` to manually set the price of ETH due to an oracle outage. However, the users or a monitoring system cannot easily follow the applied changes.

Recommendations
Short term, add events for all operations that may contribute to a higher level of monitoring and alerting.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts.

4. Incorrect GovernorshipAccepted event argument

Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-MPL-4
Target: `globals-v2/contracts/MapleGlobals.sol`	

Description
The `MapleGlobals` contract emits the `GovernorshipAccepted` event with an incorrect previous owner value.

`MapleGlobals` implements a two-step process for ownership transfer in which the current owner has to set the new governor, and then the new governor has to accept it. The `acceptGovernor` function first sets the new governor with `_setAddress` and then emits the `GovernorshipAccepted` event with the first argument defined as the old governor and the second the new one. However, because the `admin()` function returns the current value of the governor, both arguments will be the new governor.

```
function acceptGovernor() external {
     require(msg.sender == pendingGovernor, "MG:NOT_PENDING_GOVERNOR");
     _setAddress(ADMIN_SLOT, msg.sender);
     pendingGovernor = address(0);
     emit GovernorshipAccepted(admin(), msg.sender);
 }
```

Figure 4.1: `acceptGovernor` function (`globals-v2/contracts/MapleGlobals.sol#87-92`)

Exploit Scenario
The Maple team decides to transfer the `MapleGlobals`' governor to a new multi-signature wallet. The team has a script that verifies the correct execution by checking the events emitted; however, this script creates a false alert because the `GovernorshipAccepted` event does not have the expected arguments.

Recommendations
Short term, emit the `GovernorshipAccepted` event before calling `_setAddress`.

Long term, add tests to check the events have the expected arguments.

5. Partially incorrect Chainlink price feed safety checks

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-5
Target: `globals-v2/contracts/MapleGlobals.sol`	

Description

The `getLatestPrice` function retrieves a specific asset price from Chainlink. However, the price (a signed integer) is first checked that it is non-zero and then is cast to an unsigned integer with a potentially negative value. An incorrect price would temporarily affect the expected amount of fund assets during liquidation.

```
   function getLatestPrice(address asset_) external override view returns (uint256
latestPrice_) {
        // If governor has overridden price because of oracle outage, return
overridden price.
        if (manualOverridePrice[asset_] != 0) return manualOverridePrice[asset_];

        ( uint80 roundId_, int256 price_, , uint256 updatedAt_, uint80
answeredInRound_ ) =
IChainlinkAggregatorV3Like(oracleFor[asset_]).latestRoundData();

        require(updatedAt_ != 0,              "MG:GLP:ROUND_NOT_COMPLETE");
        require(answeredInRound_ >= roundId_, "MG:GLP:STALE_DATA");
        require(price_ != int256(0),          "MG:GLP:ZERO_PRICE");

        latestPrice_ = uint256(price_);
    }
```

Figure 5.1: `getLatestPrice` function (`globals-v2/contracts/MapleGlobals.sol#297-308`)

Exploit Scenario

Chainlink's oracle returns a negative value for an in-process liquidation. This value is then unsafely cast to an `uint256`. The expected amount of fund assets from the protocol is incorrect, which prevents liquidation.

Recommendations

Short term, check that the price is greater than 0.

Long term, add tests for the Chainlink price feed with various edge cases. Additionally, set up a monitoring system in the event of unexpected market failures. A Chainlink oracle can have a minimum and maximum value, and if the real price is outside of that range, it will not be possible to update the oracle; as a result, it will report an incorrect price, and it will be impossible to know this on-chain.

6. Incorrect implementation of EIP-4626

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-MPL-6
Target: `pool-v2/contracts/Pool.sol`, `PoolManager.sol`	

Description

The Pool implementation of EIP-4626 is incorrect for `maxDeposit` and `maxMint` because these functions do not consider all possible cases in which deposit or mint are disabled.

EIP-4626 is a standard for implementing tokenized vaults. In particular, it specifies the following:

- `maxDeposit`: MUST factor in both global and user-specific limits. For example, if deposits are entirely disabled (even temporarily), it MUST return 0.
- `maxMint`: MUST factor in both global and user-specific limits. For example, if mints are entirely disabled (even temporarily), it MUST return 0.

The current implementation of `maxDeposit` and `maxMint` in the Pool contract directly call and return the result of the same functions in `PoolManager` (figure 6.1). As shown in figure 6.1, both functions rely on `_getMaxAssets`, which correctly checks that the liquidity cap has not been reached and that deposits are allowed and otherwise returns 0. However, these checks are insufficient.

```
   function maxDeposit(address receiver_) external view virtual override returns
(uint256 maxAssets_) {
        maxAssets_ = _getMaxAssets(receiver_, totalAssets());
    }

    function maxMint(address receiver_) external view virtual override returns
(uint256 maxShares_) {
        uint256 totalAssets_ = totalAssets();
        uint256 totalSupply_ = IPoolLike(pool).totalSupply();
        uint256 maxAssets_   = _getMaxAssets(receiver_, totalAssets_);

        maxShares_ = totalSupply_ == 0 ? maxAssets_ : maxAssets_ * totalSupply_ /
totalAssets_;
    }

    [...]

   function _getMaxAssets(address receiver_, uint256 totalAssets_) internal view
```

```
returns (uint256 maxAssets_) {
        bool    depositAllowed_ = openToPublic || isValidLender[receiver_];
        uint256 liquidityCap_   = liquidityCap;
        maxAssets_              = liquidityCap_ > totalAssets_ && depositAllowed_ ?
liquidityCap_ - totalAssets_ : 0;
    }
```

Figure 6.1: The `maxDeposit` and `maxMint` functions (pool-v2/contracts/PoolManager.sol#L451-L461) and the `_getMaxAssets` function (pool-v2/contracts/PoolManager.sol#L516-L520)

The deposit and mint functions have a `checkCall` modifier that will call the `canCall` function in the `PoolManager` to allow or disallow the action. This modifier first checks if the global protocol pause is active; if it is not, it will perform additional checks in `_canDeposit`. For this issue, it will be impossible to deposit or mint if the Pool is not active.

```
   function canCall(bytes32 functionId_, address caller_, bytes memory data_)
external view override returns (bool canCall_, string memory errorMessage_) {
        if (IMapleGlobalsLike(globals()).protocolPaused()) {
            return (false, "PM:CC:PROTOCOL_PAUSED");
        }

        if (functionId_ == "P:deposit") {
            ( uint256 assets_, address receiver_ ) = abi.decode(data_, (uint256,
address));
            return _canDeposit(assets_, receiver_, "P:D:");
        }

        if (functionId_ == "P:depositWithPermit") {
            ( uint256 assets_, address receiver_, , , , ) = abi.decode(data_,
(uint256, address, uint256, uint8, bytes32, bytes32));
            return _canDeposit(assets_, receiver_, "P:DWP:");
        }

        if (functionId_ == "P:mint") {
            ( uint256 shares_, address receiver_ ) = abi.decode(data_, (uint256,
address));
            return _canDeposit(IPoolLike(pool).previewMint(shares_), receiver_,
"P:M:");
        }

        if (functionId_ == "P:mintWithPermit") {
            ( uint256 shares_, address receiver_, , , , , ) = abi.decode(data_,
(uint256, address, uint256, uint256, uint8, bytes32, bytes32));
            return _canDeposit(IPoolLike(pool).previewMint(shares_), receiver_,
"P:MWP:");
        }

    [...]
```

```
   function _canDeposit(uint256 assets_, address receiver_, string memory
errorPrefix_) internal view returns (bool canDeposit_, string memory errorMessage_)
{
        if (!active)                                        return (false,
_formatErrorMessage(errorPrefix_, "NOT_ACTIVE"));
        if (!openToPublic && !isValidLender[receiver_]) return (false,
_formatErrorMessage(errorPrefix_, "LENDER_NOT_ALLOWED"));
        if (assets_ + totalAssets() > liquidityCap)     return (false,
_formatErrorMessage(errorPrefix_, "DEPOSIT_GT_LIQ_CAP"));

        return (true, "");
    }
```

Figure 6.2: The `canCall` function (pool-v2/contracts/PoolManager.sol#L370-L393), and the `_canDeposit` function (pool-v2/contracts/PoolManager.sol#L498-L504)

The `maxDeposit` and `maxMint` functions should return 0 if the global protocol pause is active or if the Pool is not active; however, these cases are not considered.

Exploit Scenario

A third-party protocol wants to deposit into Maple's pool. It first calls `maxDeposit` to obtain the maximum amount of asserts it can deposit and then calls `deposit`. However, the latter function call will revert because the protocol is paused.

Recommendations

Short term, return 0 in `maxDeposit` and `maxMint` if the protocol is paused or if the pool is not active.

Long term, maintain compliance with the EIP specification being implemented (in this case, EIP-4626).

7. setAllowedSlippage and setMinRatio functions are unreachable

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MPL-7
Target: `pool-v2/contracts/LoanManager.sol`	

Description

The administrative functions `setAllowedSlippage` and `setMinRatio` have a requirement that they can be called only by the `poolManager`. However, they are not called by any reachable function in the `PoolManager` contract.

```
function setAllowedSlippage(address collateralAsset_, uint256 allowedSlippage_)
external override {
  require(msg.sender == poolManager,           "LM:SAS:NOT_POOL_MANAGER");
  require(allowedSlippage_ <= HUNDRED_PERCENT, "LM:SAS:INVALID_SLIPPAGE");

  emit AllowedSlippageSet(collateralAsset_, allowedSlippageFor[collateralAsset_] =
allowedSlippage_);
}

function setMinRatio(address collateralAsset_, uint256 minRatio_) external override
{
  require(msg.sender == poolManager, "LM:SMR:NOT_POOL_MANAGER");
  emit MinRatioSet(collateralAsset_, minRatioFor[collateralAsset_] = minRatio_);
}
```

Figure 7.1: `setAllowedSlippage` and `setMinRatio` function (`pool-v2/contracts/LoanManager.sol#L75-L85`)

Exploit Scenario

Alice, a pool administrator, needs to adjust the slippage parameter of a particular collateral token. Alice's transaction reverts since she is not the `poolManager` contract address. Alice checks the `PoolManager` contract for a method through which she can set the slippage parameter, but none exists.

Recommendations

Short term, add functions in the `PoolManager` contract that can reach `setAllowedSlippage` and `setMinRatio` on the `LoanManager` contract.

Long term, add unit tests that validate all system parameters can be updated successfully.

8. Inaccurate accounting of unrealizedLosses during default warning revert	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-MPL-8
Target: `pool-v2/contracts/LoanManager.sol`	

Description

During the process of executing the `removeDefaultWarning` function, an accounting discrepancy fails to decrement `netLateInterest` from `unrealizedLosses`, resulting in an over-inflated value.

The `triggerDefaultWarning` function updates `unrealizedLosses` with the defaulting loan's `principal_`, `netInterest_`, and `netLateInterest_` values.

```
emit UnrealizedLossesUpdated(unrealizedLosses += _uint128(principal_ + netInterest_
+ netLateInterest_));
```

Figure 8.1: The `triggerDefaultWarning` function (`pool-v2/contracts/LoanManager.sol#L331`)

When the warning is removed by the `_revertDefaultWarning` function, only the values of the defaulting loan's principal and interest are decremented from `unrealizedLosses`. This leaves a discrepancy equal to the amount of `netLateInterest_`.

```
function _revertDefaultWarning(LiquidationInfo memory liquidationInfo_) internal {
  accountedInterest -= _uint112(liquidationInfo_.interest);
  unrealizedLosses  -= _uint128(liquidationInfo_.principal +
liquidationInfo_.interest);
}
```

Figure 8.2: The `_revertDefaultWarning` function (`pool-v2/contracts/LoanManager.sol#L631-L634`)

Exploit Scenario

Alice has missed several interest payments on her loan and is about to default. Bob, the `poolManager`, calls `triggerDefaultWarning` on the loan to account for the unrealized loss in the system. Alice makes a payment to bring the loan back into good standing, the `claim` function is triggered, and `_revertDefaultWarning` is called to remove the unrealized loss from the system. The net value of Alice's loan's late interest value is still accounted for in the value of `unrealizedLosses`. From then on, when users call

`Pool.withdraw`, they will have to exchange more shares than are due for the same amount of assets.

Recommendations

Short term, add the value of `netLateInterest` to the amount decremented from `unrealizedLosses` when removing the default warning from the system.

Long term, implement robust unit-tests and fuzz tests to validate math and accounting flows throughout the system to account for any unexpected accounting discrepancies.

9. Attackers can prevent the pool manager from finishing liquidation

Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-MPL-9
Target: `pool-v2/contracts/LoanManager.sol`	

Description

The `finishCollateralLiquidation` function requires that a liquidation is no longer active. However, an attacker can prevent the liquidation from finishing by sending a minimal amount of collateral token to the liquidator address.

```
   function finishCollateralLiquidation(address loan_) external override
nonReentrant returns (uint256 remainingLosses_, uint256 platformFees_) {
        require(msg.sender == poolManager,    "LM:FCL:NOT_POOL_MANAGER");
        require(!isLiquidationActive(loan_), "LM:FCL:LIQ_STILL_ACTIVE");

        [...]

        if (toTreasury_ != 0)
ILiquidatorLike(liquidationInfo_.liquidator).pullFunds(fundsAsset, mapleTreasury(),
toTreasury_);
        if (toPool_         != 0)
ILiquidatorLike(liquidationInfo_.liquidator).pullFunds(fundsAsset, pool,
toPool_);
        if (recoveredFunds_ != 0)
ILiquidatorLike(liquidationInfo_.liquidator).pullFunds(fundsAsset,
ILoanLike(loan_).borrower(), recoveredFunds_);
```

Figure 9.1: An excerpt of the `finishCollateralLiquidation` function (`pool-v2/contracts/LoanManager.sol#L199-L232`)

The `finishCollateralLiquidation` function uses the `isLiquidationActive` function to verify if the liquidation process is finished by checking the collateral asset balance of the liquidator address. Because anyone can send tokens to that address, it is possible to make `isLiquidationActive` always return false.

```
   function isLiquidationActive(address loan_) public view override returns (bool
isActive_) {
        address liquidatorAddress_ = liquidationInfo[loan_].liquidator;

        // TODO: Investigate dust collateralAsset will ensure `isLiquidationActive`
is always true.
        isActive_ = (liquidatorAddress_ != address(0)) &&
```

```
(IERC20Like(ILoanLike(loan_).collateralAsset()).balanceOf(liquidatorAddress_) !=
uint256(0));
    }
```

Figure 9.2: The `isLiquidationActive` function (pool-v2/contracts/LoanManager.sol#L702-L707)

Exploit Scenario

Alice's loan is being liquidated. Bob, the pool manager, tries to call `finishCollateralLiquidation` to get back the recovered funds. Eve front-runs Bob's call by sending 1 token of the collateral asset to the liquidator address. As a consequence, Bob cannot recover the funds.

Recommendations

Short term, use a storage variable to track the remaining collateral in the Liquidator contract. As a result, the collateral balance cannot be manipulated through the transfer of tokens and can be safely checked in `isLiquidationActive`.

Long term, avoid using exact comparisons for ether and token balances, as users can increase those balances by executing transfers, making the comparisons evaluate to false.

10. WithdrawalManager can have an invalid exit configuration

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-10
Target: `withdrawal-manager/contracts/WithdrawalManager.sol`	

Description

The `setExitConfig` function sets the configuration to exit from the pool. However, unsafe casting allows this function to set an invalid configuration.

The function performs a few initial checks; for example, it checks that `windowDuration` is not 0 and that `windowDuration` is less than `cycleDuration`. However, when setting the configuration, the `initialCycleId_`, `initialCycleTime_`, `cycleDuration_`, and `windowDuration_` are unsafely casted to `uint64` from `uint256`. In particular, `cycleDuration_` and `windowDuration_` are user-controlled by the `poolDelegate`.

```
   function setExitConfig(uint256 cycleDuration_, uint256 windowDuration_) external
override {
        CycleConfig memory config_ = _getCurrentConfig();

        require(msg.sender == poolDelegate(),      "WM:SEC:NOT_AUTHORIZED");
        require(windowDuration_ != 0,              "WM:SEC:ZERO_WINDOW");
        require(windowDuration_ <= cycleDuration_, "WM:SEC:WINDOW_OOB");

        require(
            cycleDuration_  != config_.cycleDuration ||
            windowDuration_ != config_.windowDuration,
            "WM:SEC:IDENTICAL_CONFIG"
        );

        [...]

        cycleConfigs[latestConfigId_] = CycleConfig({
            initialCycleId:   uint64(initialCycleId_),
            initialCycleTime: uint64(initialCycleTime_),
            cycleDuration:    uint64(cycleDuration_),
            windowDuration:   uint64(windowDuration_)
        });
    }
```

Figure 10.1: The `setExitConfig` function (`withdrawal-manager/contracts/WithdrawalManager.sol#L83-L115`)

Exploit Scenario

Bob, the pool delegate, calls setExitConfig with `cycleDuration_` equal to `type(uint64).max` + 1 and `windowDuration_` equal to `type(uint64).max`. The checks pass, but the configuration does not adhere to the invariant `windowDuration <= cycleDuration`.

Recommendations

Short term, safely cast the variables when setting the configuration to avoid any possible errors.

Long term, improve the unit-tests to check that important invariants always hold.

11. Loan can be impaired when the protocol is paused

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-11
Target: `pool-v2/contracts/PoolManager.sol`	

Description

The `impairLoan` function allows the `poolDelegate` or `governor` to impair a loan when the protocol is paused due to a missing `whenProtocolNotPaused` modifier. The role of this function is to mark the loan at risk of default by updating the loan's `nextPaymentDueDate`. Although it would be impossible to default the loan in a paused state (because the `triggerDefault` function correctly has the `whenProtocolNotPaused` modifier), it is unclear if the other state variable changes would be a problem in a paused system. Additionally, if the protocol is unpaused, it is possible to call `removeLoanImpairment` and restore the loan's previous state.

```
function impairLoan(address loan_) external override {
     bool isGovernor_ = msg.sender == governor();

     require(msg.sender == poolDelegate || isGovernor_, "PM:IL:NOT_AUTHORIZED");

     ILoanManagerLike(loanManagers[loan_]).impairLoan(loan_, isGovernor_);

     emit LoanImpaired(loan_, block.timestamp);
 }
```

Figure 11.1: The `impairLoan` function (`pool-v2/contracts/PoolManager.sol#L307-315`)

Exploit Scenario

Bob, the MapleGlobal security admin, sets the protocol in a paused state due to an unknown occurrence, expecting the protocol's state to not change and debugging the possible issue. Alice, a pool delegate who does not know that the protocol is paused, calls `impairLoan`, thereby changing the state and making Bob's debugging more difficult.

Recommendations

Short term, add the missing `whenProtocolNotPaused` modifier to the `impairLoan` function.

Long term, improve the unit-tests to check for the correct system behavior when the protocol is paused and unpaused. Additionally, integrate the Slither script in appendix D into the development workflow.

12. Fee treasury could go to the zero address

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-12
Target: `pool-v2/contracts/LoanManager.sol`	

Description

The `_disburseLiquidationFunds` and `_distributeClaimedFunds` functions, which send the fees to the various actors, do not check that the `mapleTreasury` address was set.

Althoughthe `mapleTreasury` address is supposedly set immediately after the creation of the `MapleGlobals` contract, no checks prevent sending the fees to the zero address, leading to a loss for Maple.

```
   function _disburseLiquidationFunds(address loan_, uint256 recoveredFunds_,
uint256 platformFees_, uint256 remainingLosses_) internal returns (uint256
updatedRemainingLosses_, uint256 updatedPlatformFees_) {
      [...]
       require(toTreasury_     == 0 || ERC20Helper.transfer(fundsAsset_,
mapleTreasury(), toTreasury_),     "LM:DLF:TRANSFER_MT_FAILED");
```

Figure 12.1: The `_disburseLiquidationFunds` function (`pool-v2/contracts/LoanManager.sol#L566-L584`)

Exploit Scenario

Bob, a Maple admin, sets up the protocol but forgets to set the `mapleTreasury` address. Since there are no warnings, the expected claim or liquidation fees are sent to the zero address until the Maple team notices the issue.

Recommendations

Short term, add a check that the `mapleTreasury` is not set to address zero in `_disburseLiquidationFunds` and `_distributeClaimedFunds`.

Long term, improve the unit and integration tests to check that the system behaves correctly both for the happy case and the non-happy case.

Summary of Recommendations

Maple Labs's Finance platform is an advanced work in progress with multiple planned iterations. Trail of Bits recommends that Maple Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- Where possible, develop technical documentation with diagrams explaining the transaction flow for different entry-points.
- Enhance the suite of unit tests to ensure that the system behaves as expected when handling both happy and unhappy paths. This will help to identify problematic code and increase users' and developers' confidence in the code.
- Identify and analyze all system properties that are expected to hold.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Add `require(isLoanManager[loanManager_], message);` to revert with a meaningful message instead of reverting because you are trying to get an index out of bound.**

```
   function removeLoanManager(address loanManager_) external override
whenProtocolNotPaused {
        require(msg.sender == poolDelegate, "PM:RLM:NOT_PD");

        isLoanManager[loanManager_] = false;

        // Find loan manager index
        uint256 i_ = 0;
        while (loanManagerList[i_] != loanManager_) i_++;
```

Figure E.1: `removeLoanManager` function in `PoolManager` contract

- **The error message in `flashLock` modifier should be `NOT_NOT_IN_FLASH`; otherwise, it is the same as in the `onlyInFlash` modifier.**

```
    modifier onlyInFlash() {
      require(_state == IN_FLASH, "UV2S:NOT_IN_FLASH");
    [...]
    modifier flashLock() {
      require(_state == NOT_IN_FLASH, "UV2S:NOT_IN_FLASH");
```

Figure E.2: Modifiers in SushiswapStrategy/UniswapV2Strategy

- **The following error messages are incorrect:**
  - "ML:MP:TRANSFER_FAILED" should be "ML:CL:TRANSFER_FAILED"

```
   function closeLoan(uint256 amount_) external override limitDrawableUse returns
(uint256 principal_, uint256 interest_, uint256 fees_) {
      [...]
       require(ERC20Helper.transfer(_fundsAsset, _lender, principalAndInterest),
"ML:MP:TRANSFER_FAILED");
```

Figure E.3: Incorrect error message in `MapleLoan.closeLoan` function

  - "LM:TL:NOT_POOL_MANAGER" should be "LM:TD:NOT_POOL_MANAGER"

```
   function triggerDefault(address loan_) external override returns (bool
liquidationComplete_, uint256 remainingLosses_, uint256 platformFees_) {
```

```
        require(msg.sender == poolManager, "LM:TL:NOT_POOL_MANAGER");
```

Figure E.4: Incorrect error message in `MapleLoan.triggerDefault` function

- **Swap the delete operations in `acceptNewTerms` and `claim` functions to get the expected gas optimization.**

```
    function acceptNewTerms(address loan_, address refinancer_, uint256 deadline_,
bytes[] calldata calls_) external override nonReentrant {
       [...]
       delete paymentIdOf[msg.sender];
       delete payments[paymentIdOf[msg.sender]];
```

Figure E.5: Incorrect order of delete operations in `LoanManager.acceptNewTerms` function

```
   function claim(uint256 principal_, uint256 interest_, uint256
previousPaymentDueDate_, uint256 nextPaymentDueDate_) external override nonReentrant
{
       [...]
       delete paymentIdOf[msg.sender];
       delete payments[paymentIdOf[msg.sender]];
```

Figure E.6: Incorrect order of delete operations in `LoanManager.claim` function

- **Remove the unneeded approval of ERC20 token.**

```
   function _initialize(address loanManager_, address collateralAsset_, address
fundsAsset_) internal {
        require(loanManager_ != address(0), "LIQI:I:ZERO_LM");

        require(ERC20Helper.approve(collateralAsset_, loanManager_,
type(uint256).max), "LIQI:I:INVALID_C_APPROVE");
        require(ERC20Helper.approve(fundsAsset_,      loanManager_,
type(uint256).max), "LIQI:I:INVALID_F_APPROVE");
```

Figure E.7: ERC20 approval in `LiquidatorInitializer._initialize` function

D. Slither Script to Check Modifiers

Trail of Bits developed a Slither script to check for possible misuse of modifiers. The code in Figure C.1 checks that every function in `PoolManager` has the `whenProtocolNotPaused` modifier, and it works as follows:

The `_check_access_controls` function takes a modifier name and a list of functions that are expected to not have the modifier (`ACCEPTED_LIST`). Slither will iterate over each function, skipping the constructor or view functions (as these are stateless), and identify functions that do not have the modifier and are not in the accepted list of functions that do not need it.

The script can also be used with a few modifications on the `MapleGlobals` contract to check that every setter has the `isGovernor` modifier.

```
from slither import Slither
from slither.core.declarations import Contract
from typing import List

slither = Slither(".", ignore_compile=True)

def find_contract_in_compilation_units(contract_name: str) -> Contract:
    contracts = slither.get_contract_from_name(contract_name)
    return (contracts[0]) if len(contracts)>0 else print("Contract not found")

def _check_access_controls(
    contract: Contract, modifiers_access_controls: List[str], ACCEPTED_LIST:
List[str]
):
    print(f"### Check {contract} calls {modifiers_access_controls[0]}")
    no_bug_found = True
    for function in contract.functions_entry_points:
        if function.is_constructor:
            continue
        if function.view:
            continue

        if not function.modifiers or (
            not any((str(x) in modifiers_access_controls) for x in
function.modifiers)
        ):
            if not function.name in ACCEPTED_LIST:
                print(f"\t- {function.canonical_name} should have a
{modifiers_access_controls[0]} modifier")
                no_bug_found = False
    if no_bug_found:
        print("\t- No bug found")
```

```
_check_access_controls(
    find_contract_in_compilation_units("PoolManager"),
    ["whenProtocolNotPaused"], # examples: `nonreentrant` or `onlyOwner` or
`whenNotPaused`
    ["migrate", "configure"] # Functions that should not have the modifier
)
```

Figure C.1: Slither script

E. Multi-signature Wallet Best Practices

The Maple Labs team uses multi-signature wallets to perform privileged actions on the Maple protocol. This appendix summarizes the best practices for the use of a 2-of-3 multi-signature wallet where the authority to execute a transaction requires a consensus of two individuals in possession of two of the wallet's three private keys.

1. **The private keys must be stored or held separately**, and each must be respectively access-limited to separate individuals.
2. **Multiple keys should not be stored with the same custodian.** For example, if the keys are physically held in the custody of a third party (e.g., a bank), then no more than one key can be custodied in that bank. Doing so would violate best practice #1.
3. **The second signatory (a.k.a. the co-signer) must refer to a policy established beforehand** specifying the conditions for approving the transaction before signing it with their key.
4. **The co-signer should verify that the half-signed transaction was generated willfully by the intended holder of the first signature's key**.

Best practice #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first (forfeiting the decision responsibility back to the first signer, and defeating the security model). If the co-signer refuses to approve the transaction for any reason, then the due-diligence conditions for approval may be unclear. That is why a policy for validating the transaction is needed. Example verification policy rules may include:

- A predetermined protocol for being asked to cosign (e.g., a half-signed transaction will be accepted only via an approved channel).
- An allowlist of specific actions that can be performed by the wallet.
- A threshold for the number of transactions that can be executed on a given day, week, etc.

Best practice #4 mitigates the risk of a single stolen key. In a hypothetical example, an attacker somehow acquires the unlocked Ledger Nano of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction will reveal that the key has been stolen and that the transaction should not be co-signed. If under an active threat of violence, a "duress code" (code word, phrase, or other system agreed upon in advance) can be used as a covert way for one signatory to alert the others that the transaction is not being initiated willfully, without alerting the attacker.

Note that each multi-signature wallet used by the team should be treated independently of the others. Thus, each wallet should have its own policies for transaction validity and storage.

F. Incident Response Recommendations

In this section, we provide recommendations around the formulation of an incident response plan.

Identify who (either specific people or roles) is responsible for carrying out the mitigations (deploying smart contracts, pausing contracts, upgrading the front end, etc.).

- Specifying these roles will strengthen the incident response plan and ease the execution of mitigating actions when necessary.

Document internal processes for situations in which a deployed remediation does not work or introduces a new bug.

- Consider adding a fallback scenario that describes an action plan in the event of a failed remediation.

Clearly describe the intended process of contract deployment.

Consider whether and under what circumstances Maple Labs will make affected users whole after certain issues occur.

- Some scenarios to consider include an individual or aggregate loss, a loss resulting from user error, a contract flaw, and a third-party contract flaw.

Document how Maple Labs plans to keep up to date on new issues, both to inform future development and to secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.

- For each language and component, describe noteworthy sources for vulnerability news. Subscribe to updates for each source. Consider creating a special private Discord channel with a bot that will post the latest vulnerability news; this will help the team track all updates all in one place. Also consider assigning specific team members to track the vulnerability news of a specific component of the system.

Consider scenarios involving issues that would indirectly affect the system.

Determine when and how the team would reach out to and onboard external parties (auditors, affected users, other protocol developers, etc.) during an incident.

- Some issues may require collaboration with external parties to efficiently remediate them.

Define contract behavior that is considered abnormal for off-chain monitoring.

- Consider adding more resilient solutions for detection and mitigation, especially in terms of specific alternate endpoints and queries for different data as well as status pages and support contacts for affected services.

Combine issues and determine whether new detection and mitigation scenarios are needed.

Perform periodic dry runs of specific scenarios in the incident response plan to find gaps and opportunities for improvement and to develop muscle memory.

Document the intervals at which the team should perform dry runs of the various scenarios. For scenarios that are more likely to happen, perform dry runs more regularly. Create a template to be filled in after a dry run to describe the improvements that need to be made to the incident response.

G. Fix Review Results

From December 13 to December 14 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Maple Finance team for the issues identified in this report.

We reviewed each of the fixes to determine their effectiveness in resolving the associated issues. For additional information, see the Detailed Fix Log.

ID	Title	Severity	Status
1	Incorrect argument passed to _getPlatformOriginationFee	Low	Resolved
2	The protocol could stop working prematurely	Low	Resolved
3	Insufficient event generation	Low	Resolved
4	Incorrect GovernorshipAccepted event argument	Low	Resolved
5	Partially incorrect Chainlink price feed safety checks	Low	Resolved
6	Incorrect implementation of EIP-4626	Low	Unresolved
7	setAllowedSlippage and setMinRatio functions are unreachable	Low	Resolved
8	Inaccurate accounting of unrealizedLosses during default warning revert	Medium	Resolved
9	Attackers can prevent the pool manager from finishing liquidation	Medium	Resolved
10	WithdrawalManager can have an invalid exit configuration	Low	Resolved
11	Loan can be impaired when the protocol is paused	Undetermined	Resolved

12	Fee treasury could go to the zero address	Low	Resolved

Detailed Fix Log

TOB-MPL-1: Incorrect argument passed to _getPlatformOriginationFee
Resolved. The `getOriginationFees` function now passes the `loan_` argument to the `_getPlatformOriginationFee` function instead of `msg.sender`.

TOB-MPL-2: The protocol could stop working prematurely
Resolved. The `_uint48` function now uses `uint48` types instead of `uint32` types.

TOB-MPL-3: Insufficient event generation
Resolved. The suggestion that a constructor emit `DefaultTimelockParametersSet` was rendered moot by the removal of that constructor from the contract. All other suggested events were added.

TOB-MPL-4: Incorrect GovernorshipAccepted event argument
Resolved. The event, `GovernorshipAccepted`, is now emitted before the `_setAddress` function is called.

TOB-MPL-5: Partially incorrect Chainlink price feed safety checks
Resolved. The `price_` variable is now checked that it is greater than `int256(0)` to protect against zero and negative values.

TOB-MPL-6: Incorrect implementation of EIP-4626
Unresolved. Maple Finance stated:

> *Our team acknowledges that the max* functions will return non-zero values in a protocol paused scenario and will address this case in a future PoolManager release.*

TOB-MPL-7: setAllowedSlippage and setMinRatio functions are unreachable
Resolved. Functions that call `setAllowedSlippage` and `setMinRatio` were added to the `LoanManager` contract.

TOB-MPL-8: Inaccurate accounting of unrealizedLosses during default warning revert
Resolved. Maple Finance removed `netLateInterest` from the calculation of `unrealizedLosses`; therefore, it did not need to be added to the calculation in `_revertDefaultWarning`.

TOB-MPL-9: Attackers can prevent the pool manager from finishing liquidation

Resolved. Maple Finance is no longer querying the token balance by directly calling `balanceOf` on the collateral token contract. Instead, the `collateralRemaining` amount is being tracked and queried in the `Liquidator` contract.

TOB-MPL-10: WithdrawalManager can have an invalid exit configuration
Resolved. Maple Finance refactored variable casting here to use a custom method that checks that the input is within the expected range.

TOB-MPL-11: Loan can be impaired when the protocol is paused
Resolved. The `whenProtocolNotPaused` modifier was added to the `impairLoan` function.

TOB-MPL-12: Fee treasury could go to the zero address
Resolved. A check was added that ensures the `mapleTreasury_` is not equal to the zero address.