Maple Labs

Security Assessment

March 14, 2022

Prepared for:

Dan Garay
Maple Labs

Lucas Manuel
Maple Labs

Joe Flanagan
Maple Labs

Prepared by:

Maximilian Krüger, Michael Colburn, and Paweł Płatek

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 80+ 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 mutually 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 such, 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.

Table of Contents

Executive Summary

Overview

Maple Labs engaged Trail of Bits to review the security of its smart contracts. From November 15 to November 24, 2021, a team of three consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

We focused our testing efforts on the identification of flaws that could result in a compromise or lapse of confidentiality, integrity, or availability of the target system. We performed automated testing and a manual review of the code.

We achieved good coverage of each function in the codebase. However, time constraints prevented us from comprehensively reviewing and fuzzing the codebase's arithmetic or assessing all high-level loan-related scenarios and their susceptibility to sophisticated attacks. Now that we are more familiar with the codebase, we recommend that Maple Labs consider having those aspects of the project audited in a tightly scoped follow-on engagement or at least perform an additional review of them.

Summary of Findings

Most aspects of the codebase are of high quality, and the code is easy to follow. It is also clear that Maple Labs devoted significant efforts to developing documentation and testing.

Our review resulted in nine findings ranging from high to informational severity, including one of high severity and one of undetermined severity.

The project is indicative of a coding style that favors optimized and minimal code. However, certain stylistic choices resulted in deviations from security best practices. For example, the code lacks reentrancy guards, depends on the front end for zero-value checks, and includes functions that return `false` instead of reverting on a failure.

We understand that these choices were in many cases deliberate and recognize the validity of different philosophical approaches. Our views, though, are informed by our experience in auditing a wide range of projects and the security track records of those projects.

Our recommendations stem from this privileged, if not unique, vantage point. Additionally, having observed hacks resulting in significant losses, we strongly believe that code designed to handle millions of dollars in funds should start with a secure base and include exhaustive data validation and reentrancy protections, even if that leads to redundancy and verbosity. Implementing such measures rarely costs more than a couple of cents in transaction fees.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	3
Low	3
Informational	1
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Denial of Service	1
Data Validation	4
Access Controls	1
Timing	1
Undefined Behavior	2

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:

Maximilian Krüger, Consultant
max.kruger@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Project Timeline

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

Date	Event
November 10, 2021	Pre-project kickoff call
November 19, 2021	Status update meeting #1
November 30, 2021	Delivery of report draft
November 30, 2021	Report readout meeting
November 30, 2021	Addition of Fix Log (Appendix E)
December 20, 2021	Additional follow-up review
December 28, 2021	Delivery of final report

Project Targets

The engagement involved a review and testing of the targets listed below.

Debt Locker

Repository	`https://github.com/maple-labs/debt-locker`
Version	`05b4f2fe119e2ddf3dc0e441055c602f748e7d52`
Type	Solidity
Platform	Ethereum

ERC20 Helper

Repository	`https://github.com/maple-labs/erc20-helper`
Version	`60345255086d6af06c92fef17da446f3006edc14`
Type	Solidity
Platform	Ethereum

Liquidations

Repository	`https://github.com/maple-labs/liquidations`
Version	`35c628e5ab45fbffaab7aef43a030a98b712a94a`
Type	Solidity
Platform	Ethereum

Loan

Repository	`https://github.com/maple-labs/loan`
Version	`e15abbad2e1e7c675acc63db90286f85fe54515f`
Type	Solidity
Platform	Ethereum

Maple Proxy Factory

Repository	`https://github.com/maple-labs/maple-proxy-factory`
Version	`ba4475d94fc98a9ab269039dba5fe64ceba6ddda`
Type	Solidity
Platform	Ethereum

Proxy Factory

Repository	`https://github.com/maple-labs/proxy-factory`
Version	`0b5968b9d7def2426b4f84cc08e9c77b79e26f1c`
Type	Solidity
Platform	Ethereum

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.

A rating of "strong" for any one code maturity category generally requires a proactive approach to security that exceeds industry standards. We did not find the in-scope components to meet that criteria.

Category	Summary	Result
Access Controls	We found one low-severity issue concerning the lack of a two-step process for critical operations (TOB-MAPLE-003). In general, though, the access controls are satisfactory.	Satisfactory
Data Validation	We found several issues related to missing checks (TOB-MAPLE-005, TOB-MAPLE-006, TOB-MAPLE–007, TOB-MAPLE-008, TOB-MAPLE-009). The Maple Labs team indicated that it chose to omit these checks, in some cases because of its coding style and in others to maintain separation of concerns, because other components already perform those checks. Certain components lack thorough data validation and rely on other components to perform checks; this increases the likelihood that critical bugs will be introduced when code is modified. For example, the `setBorrower` function assumes that validation is taken care of by the front end; however, if there were a bug in the front end, `setBorrower` could be called with the zero value, locking the contract and preventing access to its funds (TOB-MAPLE-008). Similarly, having `ERC20Helper.transfer` revert on a failure would make it impossible to forget to check its return value, which could lead to unexpected behavior (TOB-MAPLE-006).	Weak

Arithmetic	The project uses Solidity 0.8's safe math. However, there is no invariant testing of the arithmetic.	**Satisfactory**
Assembly Use/Low-Level Calls	The use of assembly is limited and justified, and assembly is not used for optimization. However, the codebase would benefit from inline comments on the use of assembly. Additionally, the codebase contains numerous low-level calls, none of which include a contract existence check (TOB-MAPLE-007). Without these checks, low-level calls to externally owned accounts will always succeed.	**Moderate**
Code Stability	We reviewed only one set of commits of the codebase, but the codebase was still undergoing changes during the audit.	**Moderate**
Decentralization	The governor multisig contract can control many system parameters but does not have ultimate power. For example, it can propose new implementations of upgradeable loan contracts, but borrowers can decide whether to implement an upgrade. A "satisfactory" rating would require that the powers of the multisig contract be removed or given to a decentralized autonomous organization.	**Moderate**
Upgradeability	The project uses a custom implementation of the proxy pattern of upgradeability, which enables governance to propose new implementations. Contract owners can then decide whether to upgrade their contracts. With the exception of the lack of contract existence checks (TOB-MAPLE-007), we did not find any problems with the implementation.	**Satisfactory**
Function Composition	The functions are generally small and have clear purposes.	**Satisfactory**
Front-Running	The code lacks robust front-running protections. We found several issues related to front-running, which could be used to prevent lenders from performing	**Moderate**

	certain actions (TOB-MAPLE-001) or to cause users to lose their funds (TOB-MAPLE-005, TOB-MAPLE-009).	
Key Management	The Maple Labs team indicated that the private keys for the admin multisig are stored in hardware wallets.	**Satisfactory**
Monitoring	All important state-changing operations emit events. This makes it easy to monitor the state of the system. The Maple Labs team indicated that it uses Defender for event monitoring and has developed an incident response plan.	**Satisfactory**
Specification	The protocol has comprehensive documentation in the form of flow diagrams and wiki entries. All functions in the interfaces have docstrings. However, the codebase would benefit from additional inline comments. We also found one discrepancy between the implementation and the documentation, which could cause unexpected behavior and a loss of funds (TOB-MAPLE-005).	**Satisfactory**
Testing and Verification	The codebase contains comprehensive unit, integration, and fuzz tests. However, it would benefit from additional invariant testing.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Attackers can prevent lenders from funding or refinancing loans	Denial of Service	Low
2	Reentrancies can lead to misordered events	Timing	Low
3	Lack of two-step process for critical operations	Access Controls	Low
4	IERC20Like.decimals returns non-standard uint256	Undefined Behavior	Undetermined
5	Transfers in Liquidator.liquidatePortion can fail silently	Undefined Behavior	Medium
6	ERC20Helper's functions do not revert on a failure	Data Validation	Informational
7	Lack of contract existence checks before low-level calls	Data Validation	High
8	Missing zero checks	Data Validation	Medium
9	Lack of user-controlled limits for input amount in Liquidator.liquidatePortion	Data Validation	Medium

Detailed Findings

1. Attackers can prevent lenders from funding or refinancing loans	
Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-MAPLE-001
Target: `MapleLoan.sol`, `Refinancer.sol`	

Description

For the `MapleLoan` contract's `fundLoan` method to fund a new loan, the balance of `fundsAsset` in the contract must be equal to the requested principal.

```
// Amount funded and principal are as requested.
amount_ = _principal = _principalRequested;

// Cannot under/over fund loan, so that accounting works in context of PoolV1
require(_getUnaccountedAmount(_fundsAsset) == amount_, "MLI:FL:WRONG_FUND_AMOUNT");
```

Figure 1.1: An excerpt of the `fundLoan` function (`contracts/MapleLoanInternals.sol#240–244`)

An attacker could prevent a lender from funding a loan by making a small transfer of `fundsAsset` every time the lender tried to fund it (front-running the transaction). However, transaction fees would make the attack expensive.

A similar issue exists in the `Refinancer` contract: If the terms of a loan were changed to increase the borrowed amount, an attacker could prevent a lender from accepting the new terms by making a small transfer of `fundsAsset`. The underlying call to `increasePrincipal` from within the `acceptNewTerms` function would then cause the transaction to revert.

```
function increasePrincipal(uint256 amount_) external override {
    require(_getUnaccountedAmount(_fundsAsset) == amount_, "R:IP:WRONG_AMOUNT");
    _principal          += amount_;
    _principalRequested += amount_;
    _drawableFunds      += amount_;

    emit PrincipalIncreased(amount_);
}
```

Figure 1.2: The vulnerable method in the `Refinancer` contract (`contracts/Refinancer.sol#23–30`)

Exploit Scenario

A borrower tries to quickly increase the principal of a loan to take advantage of a short-term high-revenue opportunity. The borrower proposes new terms, and the lender tries to accept them. However, an attacker blocks the process and performs the profitable operation himself.

Recommendations

Short term, allow the lender to withdraw funds in excess of the expected value (by calling `getUnaccountedAmount(fundsAsset)`) before a loan is funded and between the proposal and acceptance of new terms. Alternatively, have `fundLoan` and `increasePrincipal` use greater-than-or-equal-to comparisons, rather than strict equality comparisons, to check whether enough tokens have been transferred to the contract; if there are excess tokens, use the same function to transfer them to the lender.

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`.

2. Reentrancies can lead to misordered events

Severity: **Low**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-MAPLE-002
Target: `DebtLocker.sol`, `Liquidator.sol`, `MapleLoan.sol`	

Description

Several functions in the codebase do not use the checks-effects-interactions pattern, lack reentrancy guards, or emit events after interactions. These functions interact with external and third-party contracts that can execute callbacks and call the functions again (reentering them). The event for a reentrant call will be emitted before the event for the first call, meaning that off-chain event monitors will observe incorrectly ordered events.

```
function liquidatePortion(uint256 swapAmount_, bytes calldata data_) external
override {
    ERC20Helper.transfer(collateralAsset, msg.sender, swapAmount_);

    msg.sender.call(data_);

    uint256 returnAmount = getExpectedAmount(swapAmount_);

    require(ERC20Helper.transferFrom(fundsAsset, msg.sender, destination,
returnAmount), "LIQ:LP:TRANSFER_FROM");

    emit PortionLiquidated(swapAmount_, returnAmount);
}
```

Figure 2.1: The `liquidatePortion` function (`contracts/Liquidator.sol#41–51`)

We identified this issue in the following functions:

- `DebtLocker`
  - `setAuctioneer`
  - `_handleClaim`
  - `_handleClaimOfReposessed`
  - `acceptNewTerms`
- `Liquidator`
  - `liquidatePortion`
  - `pullFunds`
- `MapleLoan`

    - `acceptNewTerms`
    - `closeLoan`
    - `fundLoan`
    - `makePayment`
    - `postCollateral`
    - `returnFunds`
    - `skim`
    - `upgrade`

Exploit Scenario

Alice calls `Liquidator.liquidatePortion` (figure 2.1). Since `fundsAsset` is an ERC777 token (or another token that allows callbacks), a callback function that Alice has registered on `ERC20Helper.transfer` is called. Alice calls `Liquidator.liquidatePortion` again from within that callback function. The event for the second liquidation is emitted before the event for the first liquidation. As a result, the events observed by off-chain event monitors are incorrectly ordered.

Recommendations

Short term, follow the checks-effects-interactions pattern and ensure that all functions emit events before interacting with other contracts that may allow reentrancies.

Long term, integrate Slither into the CI pipeline. Slither can detect low-severity reentrancies like those mentioned in this finding as well as high-severity reentrancies. Use reentrancy guards on all functions that interact with other contracts.

3. Lack of two-step process for critical operations	
Severity: **Low**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-MAPLE-003
Target: `MapleLoan.sol`	

Description

The `MapleLoan` contract's `setBorrower` and `setLender` functions transfer the privileged borrower and lender roles to new addresses. If, because of a bug or a mistake, one of those functions is called with an address inaccessible to the Maple Labs team, the transferred role will be permanently inaccessible. It may be possible to restore access to the lender role by upgrading the loan contract to a new implementation. However, only the borrower can upgrade a loan contract, so no such bailout option exists for a transfer of the borrower role to an inaccessible address. Using a two-step process for role transfers would prevent such issues.

Exploit Scenario

Alice, the borrower of a Maple loan, notices that her borrower address key might have been compromised. To be safe, she calls `MapleLoan.setBorrower` with a new address. Because of a bug in the script that she uses to set the new borrower, the new borrower is set to an address for which Alice does not have the private key. As a result, she is no longer able to access her loan contract.

Recommendations

Short term, perform role transfers through a two-step process in which the borrower or lender proposes a new address and the transfer is completed once the new address has executed a call to accept the role.

Long term, investigate whether implementing additional two-step processes could prevent any other accidental lockouts.

4. IERC20Like.decimals returns non-standard uint256	
Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-MAPLE-004
Target: `Interfaces.sol`	

Description

`IERC20Like.decimals` declares `uint256` as its return type, whereas the ERC20 standard specifies that it must return a `uint8`. As a result, functions that use the `IERC20Like` interface interpret the values returned by `decimals` as `uint256` values; this can cause values greater than 255 to enter the protocol, which could lead to undefined behavior. If the return type were `uint8`, only the last byte of the return value would be used.

Exploit Scenario

A non-standard token with a `decimals` function that returns values greater than 255 is integrated into the protocol. The code is not prepared to handle `decimals` values greater than 255. As a result of the large value, the arithmetic becomes unstable, enabling an attacker to drain funds from the protocol.

Recommendations

Short term, change the return type of `IERC20.decimals` to `uint8`.

Long term, ensure that all interactions with ERC20 tokens follow the standard.

5. Transfers in Liquidator.liquidatePortion can fail silently

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MAPLE-005
Target: `Liquidator.sol`	

Description

Calls to `ERC20Helper.transfer` in the codebase are wrapped in `require` statements, except for the first such call in the `liquidatePortion` function of the `Liquidator` contract (figure 5.1). As such, a token transfer executed through this call can fail silently, meaning that `liquidatePortion` can take a user's funds without providing any collateral in return. This contravenes the expected behavior of the function and the behavior outlined in the docstring of `ILiquidator.liquidatePortion` (figure 5.2).

```
function liquidatePortion(uint256 swapAmount_, bytes calldata data_) external override {
    ERC20Helper.transfer(collateralAsset, msg.sender, swapAmount_);

    msg.sender.call(data_);

    uint256 returnAmount = getExpectedAmount(swapAmount_);

    require(ERC20Helper.transferFrom(fundsAsset, msg.sender, destination, returnAmount),
"LIQ:LP:TRANSFER_FROM");

    emit PortionLiquidated(swapAmount_, returnAmount);
}
```

Figure 5.1: The `liquidatePortion` function (`contracts/Liquidator.sol#41–51`)

```
* @dev   Flash loan function that:
* @dev   1. Transfers a specified amount of `collateralAsset` to `msg.sender`.
* @dev   2. Performs an arbitrary call to `msg.sender`, to trigger logic necessary to get
`fundsAsset` (e.g., AMM swap).
* @dev   3. Perfroms a `transferFrom`, taking the corresponding amount of `fundsAsset` from
the user.
* @dev   If the required amount of `fundsAsset` is not returned in step 3, the entire
transaction reverts.
* @param swapAmount_ Amount of `collateralAsset` that is to be borrowed in the flashloan.
* @param data_       ABI-encoded arguments to be used in the low-level call to perform step
2.
```

```
*/
```

Figure 5.2: Docstring of `liquidatePortion` (`contracts/interfaces/ILiquidator.sol#76–83`)

Exploit Scenario

A loan is liquidated, and its liquidator contract has a collateral balance of 300 ether. The current ether price is 4,200 USDC.

Alice wants to profit off of the liquidation by taking out a flash loan of 300 ether. Having checked that the contract holds enough collateral to cover the transaction, she calls `liquidatePortion(1260000, …)` in the liquidator contract.

At the same time, Bob decides to buy 10 ether from the liquidator contract. Bob calls `Liquidator.liquidatePortion(42000)`. Because his transaction is mined first, the liquidator does not have enough collateral to complete the transfer of collateral to Alice. As a result, the liquidator receives a transfer of 1,260,000 USDC from Alice but does not provide any ether in return, leaving her with a $1,260,000 loss.

Recommendations

Short term, wrap `ERC20Helper.transfer` in a `require` statement to ensure that a failed transfer causes the entire transaction to revert.

Long term, ensure that a failed transfer of tokens to or from a user always causes the entire transaction to revert. To do that, follow the recommendations outlined in TOB-MAPLE-006 and have the `ERC20Helper.transfer` and `ERC20Helper.transferFrom` functions revert on a failure. Ensure that all functions behave as expected, that their behavior remains predictable when transactions are reordered, and that the code does not contain any footguns or surprises.

6. ERC20Helper's functions do not revert on a failure

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MAPLE-006
Target: `ERC20Helper.sol`	

Description

The `ERC20Helper` contract's `transfer`, `transferFrom`, and `approve` functions do not revert on a failure. This makes it necessary for the developer to always check their return values. A failure to perform these checks can result in the introduction of high-severity bugs that can lead to a loss of funds.

There are no uses of `ERC20Helper.transfer` for which not reverting on a failure is the best option. Making this standard behavior the default would make the code more robust and therefore more secure by default, as it would take less additional effort to make it secure.

In the rare edge cases in which a transfer is allowed to fail or a failure status should be captured in a boolean, a `try/catch` statement can be used.

Exploit Scenario

Bob, a developer, writes a new function. He calls `ERC20Helper.transfer` but forgets to wrap the call in a `require` statement. As a result, token transfers can fail silently and lead to a loss of funds if that failure behavior is not accounted for.

Recommendations

Short term, have `ERC20Helper.transfer`, `ERC20Helper.transferFrom`, and `ERC20Helper.approve` revert on a failure.

Long term, have all functions revert on a failure instead of returning `false`. Aim to make code secure by default so that less additional work will be required to make it secure. Additionally, whenever possible, avoid using optimizations that are detrimental to security.

7. Lack of contract existence checks before low-level calls	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MAPLE-007
Target: Throughout the codebase	

Description

The `ERC20Helper` contract fills a purpose similar to that of OpenZeppelin's `SafeERC20` contract. However, while OpenZeppelin's `SafeERC20` `transfer` and `approve` functions will revert when called on an address that is not a token contract address (i.e., one with zero-length bytecode), `ERC20Helper`'s functions will appear to silently succeed without transferring or approving any tokens.

If the address of an externally owned account (EOA) is used as a token address in the protocol, all transfers to it will appear to succeed without any tokens being transferred. This will result in undefined behavior.

Contract existence checks are usually performed via the `EXTCODESIZE` opcode. Since the `EXTCODESIZE` opcode would precede a `CALL` to a token address, adding `EXTCODESIZE` would make the `CALL` a "warm" access. As a result, adding the `EXTCODESIZE` check would increase the gas cost by only a little more than 100. Assuming a high gas price of 200 gwei and a current ether price of $4,200, that equates to an additional cost of 10 cents for each call to the functions of `ERC20Helper`, which is a low price to pay for increased security.

The following functions lack contract existence checks:

- `ERC20Helper`
  - `call` in `_call`
- `ProxyFactory`
  - `call` in `_initializeInstance`
  - `call` in `_upgradeInstance` (line 66)
  - `call` in `_upgradeInstance` (line 72)
- `Proxied`
  - `delegatecall` in `_migrate`
- `Proxy`
  - `delegatecall` in `_fallback`

- `MapleLoanInternals`
  - `delegatecall` in `_acceptNewTerms`

Exploit Scenario
A token contract is destroyed. However, since all transfers of the destroyed token will succeed, all Maple protocol users can transact as though they have an unlimited balance of that token. If contract existence checks were executed before those transfers, all transfers of the destroyed token would revert.

Recommendations
Short term, add a contract existence check before each of the low-level calls mentioned above.

Long term, add contract existence checks before all low-level `CALL`s, `DELEGATECALL`s, and `STATICCALL`s. These checks are inexpensive and add an important layer of defense.

8. Missing zero checks	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-MAPLE-008
Target: `Liquidator.sol`, `MapleLoan.sol`, `MapleProxyFactory.sol`	

Description

A number of constructors and functions in the codebase do not revert if zero is passed in for a parameter that should not be set to zero.

The following parameters are not checked for the zero value:

- `Liquidator` contract
  - `constructor()`
    - `owner_`
    - `collateralAsset_`
    - `fundsAsset_`
    - `auctioneer_`
    - `destination_`
  - `setAuctioneer()`
    - `auctioneer_`
- `MapleLoan` contract
  - `setBorrower()`
    - `borrower_`
  - `setLender()`
    - `lender_`
- `MapleProxyFactory` contract
  - `constructor()`
    - `mapleGlobals_`

If zero is passed in for one of those parameters, it will render the contract unusable, leaving its funds locked (and therefore effectively lost) and necessitating an expensive redeployment. For example, if there were a bug in the front end, `MapleLoan.setBorrower` could be called with `address(0)`, rendering the contract unusable and locking its funds in it.

The gas cost of checking a parameter for the zero value is negligible. Since the parameter is usually already on the stack, a zero check consists of a `DUP` opcode (3 gas) and an `ISZERO` opcode (3 gas). Given a high gas price of 200 gwei and an ether price of $4,200, a zero check would cost half a cent.

Exploit Scenario

A new version of the front end is deployed. A borrower suspects that the address currently used for his or her loan might have been compromised. As a precautionary measure, the borrower decides to transfer ownership of the loan to a new address. However, the new version of the front end contains a bug: the value of an uninitialized variable is used to construct the transaction. As a result, the borrower loses access to the loan contract, and to the collateral, forever. If zero checks had been in place, the transaction would have reverted instead.

Recommendations

Short term, add zero checks for the parameters mentioned above and for all other parameters for which zero is not an acceptable value.

Long term, comprehensively validate all parameters. Avoid relying solely on the validation performed by front-end code, scripts, or other contracts, as a bug in any of those components could prevent it from performing that validation. Additionally, integrate Slither into the CI pipeline to automatically detect functions that lack zero checks.

9. Lack of user-controlled limits for input amount in Liquidator.liquidatePortion

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MAPLE-009
Target: `Liquidator.sol`	

Description

The `liquidatePortion` function of the `Liquidator` contract computes the amount of funds that will be transferred from the caller to the liquidator contract. The computation uses an asset price retrieved from an oracle.

There is no guarantee that the amount paid by the caller will correspond to the current market price, as a transaction that updates the price feed could be mined before the call to `liquidatePortion` in the liquidator contract. EOAs that call the function cannot predict the return value of the oracle. If the caller is a contract, though, it can check the return value, with some effort.

Adding an upper limit to the amount paid by the caller would enable the caller to explicitly state his or her assumptions about the execution of the contract and to avoid paying too much. It would also provide additional protection against the misreporting of oracle prices. Since such a scenario is unlikely, we set the difficulty level of this finding to high.

Using caller-controlled limits for the amount of a transfer is a best practice commonly employed by large DeFi protocols such as Uniswap.

Exploit Scenario

Alice calls `liquidatePortion` in the liquidator contract. Due to an oracle malfunction, the amount of her transfer to the liquidator contract is much higher than the amount she would pay for the collateral on another market.

Recommendations

Short term, introduce a `maxReturnAmount` parameter and add a `require` statement—`require(returnAmount <= maxReturnAmount)`—to enforce that parameter.

Long term, always allow the caller to control the amount of a transfer. This is especially important for transfer amounts that depend on factors that can change between transactions. Enable the caller to add a lower limit for a transfer from a contract and an upper limit for a transfer of the caller's funds to a contract.

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 of users 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	Breach of the confidentiality or integrity of data
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	System failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions, locking, 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 or defense in depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the client has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

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

Categories	Description
Access Controls	The authentication and authorization of components
Arithmetic	The proper use of mathematical operations and semantics
Assembly Use	The use of inline assembly
Centralization	The existence of a single point of failure
Upgradeability	Contract upgradeability
Function Composition	The separation of the logic into functions with clear purposes
Front-Running	Resistance to front-running
Key Management	The existence of proper procedures for key generation, distribution, and access
Monitoring	The use of events and monitoring procedures
Specification	The comprehensiveness and readability of codebase documentation and specification
Testing and Verification	The use of testing techniques (e.g., unit tests and fuzzing)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found. Additionally, the component is indicative of a proactive approach to security that exceeds industry standards.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.
Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. See `crytic/building-secure-contracts` for an up-to-date version of the checklist.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

D. 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.

General Recommendations

- A number of statements in the codebase are structured such that they are concise but difficult to read. These include that in the figure below (`contracts/ProxyFactory.sol#38–40`).

```
success_ =
    implementation != address(0) &&
    _initializeInstance(proxy_ = address(new Proxy{ salt: salt_ }(address(this),
implementation)), version_, arguments_);
```

  This statement uses short circuiting. It also uses the return value of an assignment, which always makes code more difficult to follow and should be avoided.

  Consider expanding such dense statements across additional lines, which will make it easier to parse and comprehend them. For example, use a statement such as the following:

```
if (implementation != address(0)) {
    return false;
} else {
    proxy_ = new Proxy{ salt: salt_ }(address(this), implementation);
    return _initializeInstance(proxy_, version_, arguments_);
}
```

`Liquidator.sol`

- Consider having the `liquidatePortion` function return the `returnAmount` as a convenience for the caller.

E. Fix Log

On November 19, 2021, Trail of Bits reviewed the fixes and mitigations implemented by the Maple Labs team for the issues identified in this report. The Maple Labs team fixed seven of the issues reported in the original assessment, partially fixed one more, and acknowledged but did not fix the other one. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the Detailed Fix Log.

On December 20, 2021, Trail of Bits reviewed other post-audit changes introduced in the `v1.0.0` tagged commits in the `erc20-helper`, `proxy-factory`, `maple-proxy-factory`, and `liquidations` repositories and the `v2.0.0` tagged commits in the `debt-locker` and `loan` repositories. Due to the time constraints, this review focused solely on the code changes made since the commits noted in the Project Targets section. No additional issues were identified.

ID	Title	Severity	Fix Status
1	Attackers can prevent lenders from funding or refinancing loans	Low	Fixed (PR-89)
2	Reentrancies can lead to misordered events	Low	Partially Fixed (PR-9, PR-29, PR-12, PR-16)
3	Lack of two-step process for critical operations	Low	Fixed (PR-88)
4	IERC20Like.decimals returns non-standard uint256	Undetermined	Fixed (PR-28, PR-39)
5	Transfers in Liquidator.liquidatePortion can fail silently	Medium	Fixed (PR-8)
6	ERC20Helper's functions do not revert on a failure	Informational	Risk Accepted by Client

7	Lack of contract existence checks before low-level calls	High	Fixed (PR-4, PR-27, PR-90, PR-29)
8	Missing zero checks	Medium	Fixed (PR-93, PR-88, PR-15)
9	Lack of user-controlled limits for input amount in Liquidator.liquidatePortion	Medium	Fixed (PR-10)

Detailed Fix Log

TOB-MAPLE-001: Attackers can prevent lenders from funding or refinancing loans
Fixed. The Maple Labs team changed the problematic exact equality comparisons to greater-than-or-equal-to comparisons. As a result, an attacker can no longer block the funding of a loan by sending a small number of tokens to the contract. (`PR-89`)

TOB-MAPLE-002: Reentrancies can lead to misordered events
Partially fixed. The Maple Labs team changed the functions `Liquidator.pullFunds`, `DebtLocker.setAuctioneer`, `DebtLocker._handleClaim`, and `DebtLocker._handleClaimOfReposessed` to follow the checks-effects-interactions pattern. The team also added a reentrancy guard to `Liquidator.liquidatePortion` and modified the `DebtLocker.acceptNewTerms` function, though it still does not follow the checks-effects-interactions pattern. The `MapleLoan` functions mentioned in the finding remain unchanged and do not follow the checks-effects-interactions pattern. (`PR-9`, `PR-29`, `PR-12`, `PR-16`)

TOB-MAPLE-003: Lack of two-step process for critical operations
Fixed. The Maple Labs team replaced the one-step role transfer processes performed by the `setBorrower` and `setLender` functions with two-step processes executed through calls to `setProposedBorrower/acceptBorrower` and `setProposedLender/acceptLender`, respectively. (`PR-88`)

TOB-MAPLE-004: IERC20Like.decimals returns non-standard uint256
Fixed. `IERC20Like.decimals` now returns a `uint8` value, as specified in the ERC20 standard. Additionally, the Maple Labs team now requires that the `DebtLocker`'s funds and collateral assets be whitelisted when the `DebtLocker` is initialized. (`PR-28`, `PR-39`)

TOB-MAPLE-005: Transfers in Liquidator.liquidatePortion can fail silently
Fixed. The Maple Labs team wrapped the call to `ERC20Helper.transfer` in the `liquidatePortion` function in a `require` statement. As such, the entire transaction will revert if the transfer fails. (`PR-8`)

TOB-MAPLE-006: ERC20Helper's functions do not revert on a failure
Not fixed. The Maple Labs team decided not to address the issue.

TOB-MAPLE-007: Lack of contract existence checks before low-level calls
Fixed. The Maple Labs team added contract existence checks to the functions mentioned in the finding (`ERC20Helper._call`, `ProxyFactory._initializeInstance`, `Proxy._fallback`, `ProxyFactory._upgradeInstance`, `Proxied._migrate`, and `MapleLoanInternals._acceptNewTerms`). The Maple Labs team also added contract

existence checks to `ProxyFactory._registerImplementation` and `ProxyFactory._registerMigrator`. (`PR-4`, `PR-27`, `PR-90`, `PR-29`)

TOB-MAPLE-008: Missing zero checks
Fixed. The Maple Labs team added zero checks to `MapleLoanFactory.constructor` and `Liquidator.constructor`. Due to the introduction of two step-processes as a fix for TOB-MAPLE-003, zero checks for `MapleLoan`’s `setBorrower` and `setLender` functions are no longer necessary. (`PR-93`, `PR-88`, `PR-15`)

TOB-MAPLE-009: Lack of user-controlled limits for input amount in Liquidator.liquidatePortion
Fixed. The Maple Labs team added a `maxReturnAmount` parameter and a corresponding check to `Liquidator.liquidatePortion`. (`PR-10`)