Maple Labs

Security Assessment

March 14, 2022

Prepared for:

Lucas Manuel
Maple Labs

Prepared by:
Simone Monica and Justin Jacob

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

This report is considered by Trail of Bits to be public information; it is licensed to Maple Labs under the terms of the project statement of work and has been made public at Maple Labs's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

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 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 March 7 to March 11, 2022, a team of two consultants conducted a security review of the client-provided source code, with one person-week 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 target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit did not uncover any significant flaws or defects that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	0
Low	3
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	4
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:

Simone Monica, Consultant
simone.monica@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Project Timeline

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

Date	Event
March 3, 2022	Pre-project kickoff call
March 14, 2022	Delivery of report draft
March 14, 2022	Report readout meeting
March 28, 2022	Addition of fix log (appendix F)
April 12, 2022	Delivery of final report

Project Goals

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

- Are there appropriate access controls in place for user and admin operations?
- Could an attacker trap the system?
- Are there any denial-of-service attack vectors?
- Do all functions have appropriate input validation?
- Is the system vulnerable to economic attacks?
- Could users avoid paying fees during the refinancing process?
- Is it possible to replay signatures?

Project Targets

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

ERC20

Repository	https://github.com/maple-labs/erc20
Version	`756c110ddc3c96c596a52bce43553477a19ee3aa`
Type	Solidity
Platform	Ethereum

Loan

Repository	https://github.com/maple-labs/loan
Version	`58cbe527d4bfec57d9981f9d839898de7883dc65`
Type	Solidity
Platform	Ethereum

RevenueDistributionToken

Repository	https://github.com/maple-labs/revenue-distribution-token
Version	`cb98ed180ee34dbb87f22e8d7af363ec8a95bd5a`
Type	Solidity
Platform	Ethereum

xMPL

Repository	https://github.com/maple-labs/xMPL
Version	`802f182dc3e22f51add447179469f9e443b00023`
Type	Solidity
Platform	Ethereum

DebtLocker

Repository	https://github.com/maple-labs/debt-locker
Version	Pull request #60
Type	Solidity
Platform	Ethereum

MPL Migration

Repository	https://github.com/maple-labs/mpl-migration
Version	faf36fe6dcca4fe3595a08a10c3aa2ac55a54cb7
Type	Solidity
Platform	Ethereum

Additionally, changes made between the initial audit and the following releases were reviewed:

ERC20

Release	v1.0.0
Version	08db27b058049117b0503557027833d23f9858eb

Loan

Release	v3.0.0
Version	c3d3506e8e4b220ce33dba793e04bd4cc4741851

RevenueDistributionToken

Release	v1.0.1
Version	0fb7a680861338bc10826c13f02b0a54af0f2aad

xMPL

Release	v1.0.1
Version	9604d297132503cb05d74f2998c18b07f345ecc0

DebtLocker

Release	v3.0.0
Version	2734ecaeb83ad57b4331a89fc867026d9fb75806

MPL Migration

Release	v1.0.0
Version	`2d228f23c5becbf393904a95444f85f506589e3f`

Project Coverage

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

- **Loan.** The `loan` contracts allow borrowers and lenders to perform operations to agree on loan terms, fund loans, draw down funds, and refinance loans, among others. We covered an update that changed establishment fees from an upfront payment to an ongoing payment and introduced the ability to reject refinancing terms. We performed static analysis and a manual review to test access controls, input validation, and the correctness of the new features.

- **xMPL/RevenueDistributionToken/ERC20.** These contracts implement the ERC4626 standard with a rewards vesting schedule and a custom ERC20 implementation. Additionally, the owner of these contracts can migrate the underlying asset after a 10-day timelock. We performed static analysis, dynamic analysis, and a manual review. We focused on the depositing, minting, withdrawing, and redeeming operations.

Automated Testing Results

Trail of Bits has developed three unique tools for testing smart contracts. Descriptions of these tools and details on the use of tools in this project are provided below.

- Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables.
- Echidna is a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test global invariants and possible scenarios for the RevenueDistributionToken.
- Manticore is a symbolic execution framework that can exhaustively test security properties.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode, Echidna may not randomly generate an edge case that violates a property, and Manticore may fail to complete its analysis.

We follow a consistent process to maximize the efficacy of testing security properties. When using Echidna, we generate 10,000 test cases per property; when testing with Manticore, we run the tool for a minimum of one hour. In both cases, we then manually review all results.

Our automated testing and verification focused on the following system properties:

RevenueDistributionToken global invariants. We used Echidna to test the following properties that should hold when users deposit, mint, withdraw, and redeem tokens.

Property	Tool	Result
`totalAssets` is less than or equal to the underlying asset balance of the contract.	Echidna	Passed
If `totalSupply` is greater than zero, the sum of all stakers' asset balances is equal to the value of `totalAssets` (with rounding implemented).	Echidna	Passed
`totalSupply` is less than or equal to `totalAssets`.	Echidna	Passed

If `totalSupply` is greater than zero, `convertToAssets(totalSupply)` is equal to `totalAssets` (with rounding implemented).	Echidna	**Passed**
`freeAssets` is less than or equal to `totalAssets`.	Echidna	**Passed**
The staker's `balanceOfAssets` is greater than or equal to `balanceOf`.	Echidna	**Passed**

RevenueDistributionToken depositing, minting, withdrawing, and redeeming operations. The following properties test whether the system behaves properly when users deposit, mint, withdraw, and redeem tokens.

Property	Tool	Result
Depositing, minting, withdrawing, redeeming operations that are sent with the correct preconditions always succeed.	Echidna	**Passed**
Depositing tokens decreases the underlying asset balance of the sender and increases the balance of the contract.	Echidna	**Passed**
Depositing tokens increases the sender's shares by the `previewDeposit` amount.	Echidna	**Passed**
Depositing tokens increases `totalSupply` by the number of shares the caller receives.	Echidna	**Passed**
Depositing tokens increases `freeAssets` by the number of underlying assets deposited.	Echidna	**Passed**
Depositing tokens updates the `lastUpdated` variable to the current timestamp.	Echidna	**Passed**
Minting tokens decreases the sender's underlying asset balance and increases the contract's balance by the `previewMint` amount.	Echidna	**Passed**
Minting tokens increases the sender's shares by the shares requested.	Echidna	**Passed**

Minting tokens increases `totalSupply` by the number of shares the caller receives.	Echidna	Passed
Minting tokens increases `freeAssets` by the number of underlying assets deposited.	Echidna	Passed
Minting tokens updates the `lastUpdated` variable to the current timestamp.	Echidna	Passed
Withdrawing tokens decreases the sender’s balance of shares by the `previewWithdraw` amount.	Echidna	Passed
Withdrawing tokens increases the sender’s asset balance and decreases the contract’s balance by the amount requested.	Echidna	Passed
Withdrawing tokens decreases `freeAssets` by the amount requested.	Echidna	Passed
Withdrawing tokens decreases `totalSupply` by the `previewWithdraw` amount.	Echidna	Passed
Withdrawing tokens updates the `lastUpdated` variable to the current timestamp.	Echidna	Passed
Redeeming tokens decreases the sender’s balance by the amount requested.	Echidna	Passed
Redeeming tokens increases the sender’s asset balance and decreases the contract’s balance by the `previewRedeem` amount.	Echidna	Passed
Redeeming tokens decreases `freeAssets` by the `previewRedeem` amount.	Echidna	Passed
Redeeming tokens decreases `totalSupply` by the amount requested.	Echidna	Passed
Redeeming tokens updates the `lastUpdated` variable to the current timestamp.	Echidna	Passed

When `totalSupply` is greater than zero, it is not possible to gain more assets by depositing/minting and withdrawing/redeeming in the same transaction.	Echidna	Passed
When `totalSupply` is zero, it is not possible to gain more assets by depositing/minting and withdrawing/redeeming in the same transaction.	Echidna	TOB-MPL-05

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
Arithmetic	The project uses Solidity 0.8's safe math. Moreover, we were provided with invariants for the `xMPL` contract that the Maple Labs team already tested.	**Satisfactory**
Auditing	Both the updated `loan` contracts and `xMPL` contract emit appropriate events for monitoring the system. Additionally, the Maple Labs team indicated that it uses Defender for event monitoring and has developed an incident response plan.	**Satisfactory**
Authentication / Access Controls	Appropriate access controls are in place for the updated `loan` contracts. The `xMPL` contract has a single privileged actor for whom access controls are in place.	**Satisfactory**
Complexity Management	The functionalities added to the `loan` contracts are small and easy to understand. The `xMPL` contract's functions are well separated and documented. However, we found some functions that would benefit from additional data validation (TOB-MPL-01, TOB-MPL-03, TOB-MPL-04, TOB-MPL-06).	**Moderate**
Cryptography and Key Management	The Maple Labs team indicated that the private keys for the admin multisignature wallet are stored in hardware wallets.	**Satisfactory**

Decentralization	Since the last audit, no significant changes were made to the `loan` contracts in terms of upgradeability and decentralization. The `xMPL` contract has a privileged actor who can migrate the underlying asset after a 10-day timelock.	**Moderate**
Documentation	The protocol has comprehensive documentation in the form of flow diagrams and wiki entries. All functions in the interfaces have docstrings. However, the migration process in the xMPL contract could be documented better.	**Satisfactory**
Front-Running Resistance	We found one issue related to missing protection against the ERC20 `approve` race condition (TOB-MPL-02); however, the updates to the `loan` contracts do not introduce possible problematic functions.	**Satisfactory**
Low-Level Calls	Low-level calls are minimal and have the necessary safeguards.	**Satisfactory**
Testing and Verification	The codebase contains adequate unit tests. Additionally, fuzz testing is used to test the system's invariants.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Risk of reuse of signatures across forks due to lack of chain ID validation	Data Validation	**High**
2	Risk of token theft due to race condition in ERC20's approve function	Timing	**High**
3	Missing check on newAsset's decimals	Data Validation	**Low**
4	Lack of zero address checks	Data Validation	**Low**
5	Possibility that users could receive more assets than the amount due	Undefined Behavior	**Low**
6	Signature malleability due to use of ecrecover	Data Validation	**Informational**
7	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**

Detailed Findings

1. Risk of reuse of signatures across forks due to lack of chain ID validation	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-01
Target: `erc20/contracts/ERC20Permit.sol`	

Description

The `ERC20Permit` contract implements EIP-2612 functionality, in which a domain separator containing the chain ID is included in the signature schema. However, the chain ID is fixed at the time of deployment. In the event of a post-deployment hard fork of the chain, the chain ID cannot be updated, and signatures may be replayed across both versions of the chain. If a change in the chain ID is detected, the domain separator can be cached and regenerated.

Exploit Scenario

Bob holds tokens worth $1,000 on the mainnet. Bob submits a signature to permit Eve to spend those tokens on his behalf. Later, the mainnet is hard-forked and retains the same chain ID. As a result, there are two parallel chains with the same chain ID, and Eve can use Bob's signature to transfer funds on both chains.

Recommendations

Short term, to prevent post-deployment forks from affecting calls to `permit`, add code to `permit` that checks `block.chainId` against `chainId` and recomputes the `DOMAIN_SEPARATOR` if they are different.

Long term, identify and document the risks associated with having forks of multiple chains and develop related mitigation strategies.

2. Risk of token theft due to race condition in ERC20's approve function	
Severity: **High**	Difficulty: **High**
Type: Timing	Finding ID: TOB-MPL-02
Target: `erc20/contracts/ERC20.sol`, `ERC20Permit.sol`	

Description

A known race condition in the ERC20 standard's `approve` function could lead to token theft.

The ERC20 standard describes how to create generic token contracts. Among others, an ERC20 contract defines these two functions:

- `transferFrom(from, to, value)`
- `approve(spender, value)`

These functions can be called to give permission to a third party to spend tokens. When a user calls the `approve(spender, value)` function, the `spender` can spend up to the `value` of the caller's tokens by calling `transferFrom(user, to, value)`.

This schema is vulnerable to a race condition in which the user calls `approve` a second time on a `spender` that has already been approved. Before the second transaction is mined, the `spender` can call `transferFrom` to transfer the previously approved value and still receive the authorization to transfer the new approved value.

Exploit Scenario

Alice calls `approve(Bob, 1000)`, allowing Bob to spend 1,000 tokens. Alice changes her mind and calls `approve(Bob, 500)`. Once mined, this transaction will decrease the number of tokens that Bob can spend to 500. Bob sees the second transaction—`approve(Bob, 500)`—and calls `transferFrom(Alice, X, 1000)` before it is mined. Bob's transaction is mined before Alice's, and Bob transfers 1,000 tokens. Once Alice's transaction is mined, Bob calls `transferFrom(Alice, X, 500)`. Essentially, Bob is able to transfer 1,500 tokens even though Alice intended that he be able to transfer only 500.

Recommendations

Short term, add two non-ERC20 functions allowing users to increase and decrease the approval (`increaseAllowance`, `decreaseAllowance`).

Long term, when implementing custom ERC20 contracts, use `slither-check-erc` to check that the contracts adhere to the specification and are protected against this issue.

3. Missing check on newAsset’s decimals

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-03
Target: `mpl-migrator/contracts/Migrator.sol`	

Description

The `Migrator` contract allows users to migrate their MPL tokens to a new version of the token; however, it lacks a check to ensure that `newAsset`'s decimals are equal to the old asset's decimals.

A migration functionality is implemented in the `xMPL` contract, allowing users who deposited their MPL tokens to easily migrate them to the new version.

```
constructor(address oldToken_, address newToken_) {
     oldToken = oldToken_;
     newToken = newToken_;
 }
```

Figure 3.1: `Migrator.sol#L11-L14`

Exploit Scenario

Bob, the owner of `xMPL`, calls `performMigration` to migrate the underlying asset to the new version, which has different decimals. Alice, a Maple user, decides to redeem her shares after the migration, and she receives an incorrect amount due to the different decimals on the new asset.

Recommendations

Short term, in the `Migrator` contract's constructor, add a check to verify that `newAsset`'s decimals are equal to the old asset's decimals.

Long term, when implementing a migration of a component, make sure that checks are in place to verify the correctness of all data related to the migrated component.

4. Lack of zero address checks

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-04
Target: `erc20/contracts/ERC20Permit.sol`	

Description

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

The following parameters do not have zero address checks:

- The `owner_` and `spender_` parameters of the `_approve` function
- The `owner_` and `recipient_` parameters of the `_transfer` function
- The `recipient_` parameter of the `_mint` function
- The `owner_` parameter of the `_burn` function

Exploit Scenario

Alice, a user of the `xMPL` contract, tries to send 100 xMPL tokens; however, she does not set the recipient, and her wallet incorrectly validates it as the zero address. She loses her tokens.

Recommendations

Short term, add zero address checks for the parameters listed 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 them from performing that validation.

5. Possibility that users could receive more assets than the amount due	
Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-MPL-05
Target: `contracts/RevenueDistributionToken.sol`	

Description

If `totalSupply` is zero (i.e., no one has deposited yet), the first user who deposits after `updateVestingSchedule` is called could immediately redeem his tokens to get back more of the asset than the amount he deposited.

This is possible because, by design, when `totalSupply` is zero, the number of shares minted corresponds to the number of assets deposited.

```
   function convertToShares(uint256 assets_) public view override returns (uint256
shares_)    {
        uint256 supply = totalSupply;  // Cache to memory.

        shares_ = supply == 0 ? assets_ : (assets_ * supply) / totalAssets();
    }
```

Figure 5.1: `RevenueDistributionToken.sol#L190-L195`

Exploit Scenario

Bob, the owner of the `xMPL` contract, decides to deposit rewards. He calls `updateVestingSchedule` without noticing that there are not yet any depositors. Eve deposits tokens and redeems them in the same transaction, receiving an unfair number of assets (appendix E).

Recommendations

Short term, make sure there is at least one depositor before calling `updateVestingSchedule`.

Long term, document assumptions about possible edge cases that can occur when operating the protocol.

6. Signature malleability due to use of ecrecover

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MPL-06
Target: `erc20/contracts/ERC20Permit.sol`	

Description

The `ERC20Permit` contract implements EIP-2612 functionality, which requires the use of the precompiled EVM contract `ecrecover`. This contract is susceptible to signature malleability due to non-unique s and v values, which could allow users to conduct replay attacks. However, the current implementation is protected from possible replay attacks due to its use of nonces.

```
   function permit(address owner, address spender, uint256 amount, uint256 deadline,
uint8 v, bytes32 r, bytes32 s) external override {
        require(deadline >= block.timestamp, "ERC20Permit:EXPIRED");
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, amount,
nonces[owner]++, deadline))
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress == owner && owner != address(0),
"ERC20Permit:INVALID_SIGNATURE");
        _approve(owner, spender, amount);
    }
```

Figure 6.1: `ERC20Permit.sol#L72-L84`

Recommendations

Short term, to prevent the future misuse of `ecrecover`, implement appropriate checks on the s and v values to verify that s is in the lower half of the range and v is 27 or 28.

Long term, identify and document the risks associated with the use of `ecrecover` and Maple Labs's plans to mitigate them.

7. Solidity compiler optimizations can be problematic	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-MPL-07
Target: `foundry.toml`	

Description

The Maple contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Maple contracts.

Recommendations

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

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
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
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.

`ERC20Permit.sol`

- **Replace the use of inline assembly to get the `chainId` with `block.chainId`.**

```
uint256 chainId;
assembly {
    chainId := chainid()
}
```

Figure C.1: `ERC20Permit.sol#L48-L51`

- **The balance increments in `_transfer` and `_mint` can be done inside an unchecked block to save gas.**

D. ERC4626 Conformance

Trail of Bits added support for ERC4626 in `slither-check-erc` to ensure that the `RevenueDistributionToken` contract conforms to the ERC4626 standard. It will check for the presence of the expected functions and that they return the correct type and emit the appropriate events.

```
$ slither-check-erc --erc erc4626 contracts/RevenueDistributionToken.sol
RevenueDistributionToken

# Check RevenueDistributionToken

## Check functions
[✓] asset() is present
       [✓] asset() -> (address) (correct return type)
       [✓] asset() is view
[✓] totalAssets() is present
       [✓] totalAssets() -> (uint256) (correct return type)
       [✓] totalAssets() is view
[✓] convertToShares(uint256) is present
       [✓] convertToShares(uint256) -> (uint256) (correct return type)
       [✓] convertToShares(uint256) is view
[✓] convertToAssets(uint256) is present
       [✓] convertToAssets(uint256) -> (uint256) (correct return type)
       [✓] convertToAssets(uint256) is view
[✓] maxDeposit(address) is present
       [✓] maxDeposit(address) -> (uint256) (correct return type)
       [✓] maxDeposit(address) is view
[✓] previewDeposit(uint256) is present
       [✓] previewDeposit(uint256) -> (uint256) (correct return type)
       [✓] previewDeposit(uint256) is view
[✓] deposit(uint256,address) is present
       [✓] deposit(uint256,address) -> (uint256) (correct return type)
       [✓] Deposit(address,address,uint256,uint256) is emitted
[✓] maxMint(address) is present
       [✓] maxMint(address) -> (uint256) (correct return type)
       [✓] maxMint(address) is view
[✓] previewMint(uint256) is present
       [✓] previewMint(uint256) -> (uint256) (correct return type)
       [✓] previewMint(uint256) is view
[✓] mint(uint256,address) is present
       [✓] mint(uint256,address) -> (uint256) (correct return type)
       [✓] Deposit(address,address,uint256,uint256) is emitted
[✓] maxWithdraw(address) is present
       [✓] maxWithdraw(address) -> (uint256) (correct return type)
```

```
    [✓] maxWithdraw(address) is view
[✓] previewWithdraw(uint256) is present
    [✓] previewWithdraw(uint256) -> (uint256) (correct return type)
    [✓] previewWithdraw(uint256) is view
[✓] withdraw(uint256,address,address) is present
    [✓] withdraw(uint256,address,address) -> (uint256) (correct return type)
    [✓] Withdraw(address,address,address,uint256,uint256) is emitted
[✓] maxRedeem(address) is present
    [✓] maxRedeem(address) -> (uint256) (correct return type)
    [✓] maxRedeem(address) is view
[✓] previewRedeem(uint256) is present
    [✓] previewRedeem(uint256) -> (uint256) (correct return type)
    [✓] previewRedeem(uint256) is view
[✓] redeem(uint256,address,address) is present
    [✓] redeem(uint256,address,address) -> (uint256) (correct return type)
    [✓] Withdraw(address,address,address,uint256,uint256) is emitted
[✓] totalSupply() is present
    [✓] totalSupply() -> (uint256) (correct return type)
    [✓] totalSupply() is view
[✓] balanceOf(address) is present
    [✓] balanceOf(address) -> (uint256) (correct return type)
    [✓] balanceOf(address) is view
[✓] transfer(address,uint256) is present
    [✓] transfer(address,uint256) -> (bool) (correct return type)
    [✓] Transfer(address,address,uint256) is emitted
[✓] transferFrom(address,address,uint256) is present
    [✓] transferFrom(address,address,uint256) -> (bool) (correct return type)
    [✓] Transfer(address,address,uint256) is emitted
[✓] approve(address,uint256) is present
    [✓] approve(address,uint256) -> (bool) (correct return type)
    [✓] Approval(address,address,uint256) is emitted
[✓] allowance(address,address) is present
    [✓] allowance(address,address) -> (uint256) (correct return type)
    [✓] allowance(address,address) is view
[✓] name() is present
    [✓] name() -> (string) (correct return type)
    [✓] name() is view
[✓] symbol() is present
    [✓] symbol() -> (string) (correct return type)
    [✓] symbol() is view
[✓] decimals() is present
    [✓] decimals() -> (uint8) (correct return type)
    [✓] decimals() is view

## Check events
[✓] Deposit(address,address,uint256,uint256) is present
```

```
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed
[✓] Withdraw(address,address,address,uint256,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed
        [✓] parameter 2 is indexed
```

Figure D.1: Running `slither-check-erc` on `RevenueDistributionToken`

E. Proof of Concept for TOB-MPL-05

The following is a test that reproduces the issue described in finding TOB-MPL-05.

```
contract TestFail is TestUtils {
    MockERC20 asset;
    RDT       rdToken;
    Staker    staker;
    Owner     owner;

    function setUp() public virtual {
        asset   = new MockERC20("MockToken", "MT", 18);
        owner = new Owner();
        rdToken = new RDT("Revenue Distribution Token", "RDT", address(owner),
address(asset), 1e30);
        staker  = new Staker();

        vm.warp(10_000_000);  // Warp to non-zero timestamp
    }

    function test_fail() external {
        // Update vesting schedule
        uint256 vestingAmount = 1000e18;
        uint256 vestingPeriod = 1522000;
        asset.mint(address(owner), vestingAmount);
        owner.erc20_transfer(address(asset), address(rdToken), vestingAmount);
        owner.rdToken_updateVestingSchedule(address(rdToken), vestingPeriod);

        // Update block.timestamp of 1 second
        vm.warp(10_000_001);

        // Staker Deposit
        uint256 depositAmount = 1;
        asset.mint(address(staker), depositAmount);
        staker.erc20_approve(address(asset), address(rdToken), depositAmount);
        uint256 shares = staker.rdToken_deposit(address(rdToken), depositAmount);
        assertEq(shares, rdToken.balanceOf(address(staker)));

        // Staker Redeem
        staker.rdToken_redeem(address(rdToken), 1);

        // [FAIL] test_fail() (gas: 204869)
        // Logs:
        //   Error: a == b not satisfied [uint]
        //     Expected: 1
        //       Actual: 657030223390276
        assertEq(asset.balanceOf(address(staker)), depositAmount);
    }
}
```

F. Fix Log

On March 28, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Maple Labs team for issues identified in this report. The Maple Labs team fixed five of the issues reported in the original assessment and did not fix the other two. 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.

ID	Title	Severity	Fix Status
1	Risk of reuse of signatures across forks due to lack of chain ID validation	High	Fixed (PR 23)
2	Risk of token theft due to race condition in ERC20's approve function	High	Fixed (PR 20)
3	Missing check on newAsset's decimals	Low	Fixed (PR 8)
4	Lack of zero address checks	Low	Not fixed
5	Possibility that users could receive more assets than the amount due	Low	Fixed (PR 37)
6	Signature malleability due to use of ecrecover	Informational	Fixed (PR 26)
7	Solidity compiler optimizations can be problematic	Informational	Not fixed

Detailed Fix Log

TOB-MPL-1: Risk of reuse of signatures across forks due to lack of chain ID validation
Fixed. The `DOMAIN_SEPARATOR` will now be recomputed every time the `permit` function is called.

TOB-MPL-2: Risk of token theft due to race condition in ERC20's approve function
Fixed. The Maple Labs team added the `increaseAllowance` and `decreaseAllowance` functions, which can prevent this issue.

TOB-MPL-3: Missing check on newAsset's decimals
Fixed. The `newAsset`'s decimals are now checked against the `oldAsset`'s decimals in the constructor.

TOB-MPL-4: Lack of zero address checks
Not fixed.

TOB-MPL-5: Possibility that users could receive more assets than the amount due
Fixed. The `updateVestingSchedule` function now checks that `totalSupply` is not zero, which indicates there is at least one depositor.

TOB-MPL-6: Signature malleability due to use of ecrecover
Fixed. The `permit` function now performs the appropriate checks on the s and v values.

TOB-MPL-7: Solidity compiler optimizations can be problematic
Not fixed.