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

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 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	Project pre-kickoff call
March 14, 2022	Delivery of report draft
March 14, 2022	Report readout meeting

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 DoS attack vectors?
- Do all functions have appropriate input validation?
- Are there possible economic attacks?
- Is it possible to not pay fees during the refinance 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

Revenue distribution token

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

Debt Locker

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

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 the borrower and the lender to perform operations such as: agree on loan terms, fund the loan, draw down the funds, refinance the loan. We covered the update regarding the establishment fees from an upfront payment to an ongoing payment and the ability to reject refinance terms. We performed static analysis and a manual review to test access control, input validation and the correctness of the new features.

- **xMPL/RevenueDistributionToken/ERC20.** These contracts implement the ERC4626 with a rewards' vesting schedule and a custom ERC20 implementation, it's also possible for the owner to migrate the underlying asset after a 10 days timelock. We performed static analysis, dynamic analysis, and a manual review. We focused on the deposit/mint/withdraw/redeem 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 when depositing/minting/withdrawing/redeeming.

Property	Tool	Result
The `totalAssets` is less than or equal to the underlying asset balance of the contract.	Echidna	**Passed**
If `totalSupply` is greater than 0, the sum of the balance of assets of every staker is equal to the `totalAssets` with rounding.	Echidna	**Passed**
The `totalSupply` is less than or equal to the `totalAssets`.	Echidna	**Passed**

If `totalSupply` is greater than 0, `convertToAssets(totalSupply)` is equal to `totalAssets` with rounding.	Echidna	Passed
The `freeAssets` is less than or equal to the `totalAssets`.	Echidna	Passed
The staker's `balanceOfAssets` is greater than or equal to the `balanceOf`.	Echidna	Passed

RevenueDistributionToken deposit/mint/withdraw/redeem operations. The following properties test that the system behaves properly when users deposit, mint, withdraw, redeem.

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

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

When `totalSupply` is greater than 0 it is not possible to gain more assets by depositing/minting and withdrawing/redeeming in the same transaction.	Echidna	Passed
When `totalSupply` is 0 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, for the xMPL contract, we were provided with invariants already tested by the Maple Labs team.	Satisfactory
Auditing	Both the updated Loan contract 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 contract. The xMPL has a single privileged actor for whom access controls are in place.	Satisfactory
Complexity Management	The Loan added functionalities are small and easy to understand. The xMPL contract 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	The Maple Labs team indicated that the private keys for the admin multisig are stored in hardware wallets.	Satisfactory

Management		
Decentralization	For the Loan codebase no significant changes have been done regarding the upgradeability and the decentralization since the last audit. The xMPL contract has a privileged actor who can migrate the underlying asset after a 10 days timelock.	**Moderate**
Documentation	The protocol has comprehensive documentation in the form of flow diagrams and wiki entries. All functions in the interfaces have docstrings. The xMPL migration process could be documented better.	**Satisfactory**
Front-Running Resistance	We found one issue related to the missing protection against ERC20 `approve` race condition (TOB-MPL-02), however the updated Loan protocol doesn't add possible problematic functions.	**Satisfactory**
Low-Level Calls	Low-level calls are minimal with the necessary safeguards	**Satisfactory**
Testing and Verification	The codebase contains adequate unit tests. Additionally fuzz testing is used to test 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	Lack of chain ID validation allows reuse of signatures across forks	Data Validation	**High**
2	Race condition in the ERC20 "approve" function may lead to token theft	Timing	**High**
3	Missing check on newAsset's decimals	Data Validation	**Low**
4	Lack of zero address checks	Data Validation	**Low**
5	User could receive back more assets amount than due	Undefined Behavior	**Low**
6	Use of ecrecover allows signature malleability	Data Validation	**Informational**
7	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**

Detailed Findings

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

Description

The ERC20Permit implements EIP-2612 in which a domain separator containing the chainId is included in the signature schema. However, the chainId is fixed at the time of deployment.

In the event of a post-deployment chain hard fork, the chainId cannot be updated, and signatures may be replayed across both versions of the chain. If a change in the chainId is detected, the domain separator can be cached and regenerated.

Exploit Scenario

Bob holds tokens worth $1,000 on Mainnet. Bob has submitted a signature to permit Eve to spend those tokens on his behalf. Later, Mainnet is hard-forked and retains the same chainId. As a result, there are two parallel chains with the same chainId, 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 a check in permit in which if the block.chainId is different from chainId the DOMAIN_SEPARATOR will be recomputed.

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

2. Race condition in the ERC20 "approve" function may lead to token theft	
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, on the 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 give permission to a third party to spend tokens. Once the function approve(spender, value) has been called by a user, spender can spend up to the value of the user's tokens by calling transferFrom(user, to, value).

This schema is vulnerable to a race condition, where the user calls approve a second time on a spender that has already been allowed. If the spender sees the transaction containing the call before it has been mined, the spender can call transferFrom to transfer the previous value and still receive the authorization to transfer the new value.

Exploit Scenario

Alice calls approve(Bob, 1000). This allows Bob to spend 1,000 tokens.
Alice changes her mind and calls approve(Bob, 500). Once mined, this will decrease to 500 the number of tokens that Bob can spend.
Bob sees the second transaction and calls transferFrom(Alice, X, 1000) before approve(Bob, 500) has been 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). Bob has transferred 1,500 tokens even though this was not Alice's intention.

Recommendations

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

Long term, when implementing custom ERC20 contracts use slither-check-erc to check that it adheres to the specification and it's safe 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 from the MPL token to a new version, however it lacks a check to ensure that newAsset's decimals are equal to the old asset's decimals.
Furthermore a migration functionality is implemented in the xMPL contract to easily let the users who deposited their MPL token migrate 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, after the migration happens decides to redeem her shares and she receives an apparent incorrect amount due to the different decimals on the new asset.

Recommendations

Short term, in the Migrator's constructor check that newAsset's decimals are equal to the old asset's decimals.

Long term, when implementing a migration of a component make sure to check for the correctness of every data related to the component migrated.

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 are not checked for the zero value:

- `_approve`
  - `owner_`
  - `spender_`
- `_transfer`
  - `owner_`
  - `recipient_`
- `_mint`
  - `recipient_`
- `_burn`
  - `owner_`

Exploit Scenario

Alice, a user of xMPL, tries to send 100 xMPL tokens however she doesn't set the recipient and her wallet incorrectly validates it as address zero, she loses her tokens.

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.

5. User could receive back more asset's amount than due

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-MPL-05
Target: `contracts/RevenueDistributionToken.sol`	

Description

If the `totalSupply` is 0 (i.e. no one deposited yet) the first user who deposits after `updateVestingSchedule` is called could immediately redeem to get back more asset's amount than the amount deposited.

This is possible because by design when the `totalSupply` is 0 the shares amount minted corresponds to the assets amount 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 xMPL decides to deposit rewards and call updateVestingSchedule without noticing there aren't any depositors. Eve deposits and redeems in the same transaction receiving back an unfair amount of assets (Appendix E).

Recommendations

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

Long term, document assumptions and possible edge cases when operating the protocol.

6. Use of ecrecover allows signature malleability

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

Description

The ERC20Permit implements EIP-2612 which requires the use of ecrecover. The EVM precompile ecrecover is susceptible to signature malleability which could allow replay attacks due to non unique s and v values. However the current implementation is safe from possible replay attacks as it uses 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 future incorrect use of ecrecover implement the appropriate check on the s and v values. Require that s is in the lower half and v is 27 or 28.

Long term, identify and document the risk associated with the use of ecrecover and how you plan 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 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
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 Calls	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	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. 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 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 the 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, 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 (TOB-MPL-05)

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

```
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);
    }
}
```