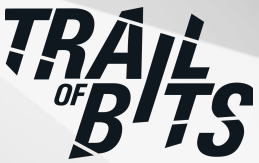

Primitive

Security Assessment

February 28, 2022

Prepared for:
Alex
Primitive

Prepared by: **Nat Chin** and **Simone Monica**

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 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

Primitive engaged Trail of Bits to review the security of its smart contracts. From January 3 to January 28, 2022, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks 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 smart contract compromise or loss of funds. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed automated testing and a manual review of the code, in addition to running system invariants.

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 and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	3
Low	1
Informational	6
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Patching	1
Timing	3
Undefined Behavior	5

Notable Findings

Findings that may impact the smart contracts are listed below.

- **TOB-PTV-1**
  Due to the lack of contract existence checks, transfer operations may silently fail. As a result, the pool may assume that failed transfers were successful, which may result in incorrect accounting.

- **TOB-PTV-6, TOB-PTV-11, TOB-PTV-12**
  These issues are related to arithmetic functions that are not direct inverses of each other, which can result in the calculation of different amounts than expected.

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:

Nat Chin, Consultant
natalie.chin@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

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

Date	Event
January 3, 2022	Pre-project kickoff call
January 10, 2022	Status update meeting #1
January 18, 2022	Status update meeting #2
January 24, 2022	Status update meeting #3
January 31, 2022	Delivery of report draft
January 31, 2022	Report readout meeting
February 4, 2022	Delivery of final report
February 16, 2022	Renaming of references from "Primitive Finance" to "Primitive"
February 28, 2022	Addition of Primitive statement on TOB-PTV-2

Project Goals

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

- Is it possible to steal funds?
- Are there appropriate access control measures in place for users and admins?
- Does the system's behavior match the specification?
- Can an attacker trap the system?
- Is it possible to perform swaps without paying the required amount?
- Are the arithmetic libraries used correctly, and do they correctly apply rounding?

Project Targets

rmm-core

Repository	https://github.com/primitivefinance/rmm-core
Version	`5dcf4306fc32fb9a4e3c154deb86f6b9d513c344`
Type	Ethereum
Platform	Solidity

rmm-manager

Repository	https://github.com/primitivefinance/rmm-manager
Version	`b0ce230a1b9752b873f3f766f671e70c59ecd6d1`
Type	Ethereum
Platform	Solidity

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:

- **`rmm-core`.** The `rmm-core` folder contains two main contracts, `PrimitiveFactory` and `PrimitiveEngine`. The `PrimitiveFactory` contract is used to deploy a `PrimitiveEngine` contract with user-specified risky and stable tokens. The `PrimitiveEngine` contract is the system's core contract in which users can create a pool with specific parameters, deposit and withdraw risky and stable tokens to and from an internal bookkeeping system, allocate and remove liquidity to and from a certain pool, and swap risky and stable tokens. We used static analysis, a manual review, and Echidna to test the behavior of these actions.

- **`rmm-manager`.** The `rmm-manager` directory contains primarily periphery contracts that are used to operate the `rmm-core` components. These contracts are the users' entry point for interacting with the system.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Fuzzing coverage on `rmm-manager`

Automated Testing Results

Trail of Bits has developed unique tools for testing smart contracts. In this assessment, we used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to check various system states.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations; for example, Echidna may not randomly generate an edge case that violates a property. We follow a consistent process to maximize the efficacy of testing security properties. When using Echidna, we generate 30,000 test cases per property.

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

Libraries. Using Echidna, we tested assumptions on the inverse of expected function behavior.

ID	Property	Tool	Result
1	The `scaleUp` and `scaleDown` functions are inverses of each other.	Echidna	**Passed**
2	The `scaleToX64` and `scalefromX64` functions are inverses of each other.	Echidna	TOB-PT V-12
3	The percentage functions are inverses of each other.	Echidna	TOB-PT V-6
4	`getCDF` always returns output in the range of `(0, 1)`.	Echidna	TOB-PT V-13
5	The delta between the error function defined in the white paper and the actual implementation is insignificant.	Echidna	**Passed**

`rmm-core` end-to-end global system checks. The following properties ensure that the system's variables cannot be changed after deployment.

ID	Property	Tool	Result
6	When the `PrimitiveEngine` contract is deployed, the `PRECISION()` constant is `10**18`.	Echidna	**Passed**
7	When the `PrimitiveEngine` contract is deployed, the `MIN_LIQUIDITY()` constant is greater than zero.	Echidna	**Passed**
8	When the `PrimitiveEngine` contract is deployed, the engine's risky and stable tokens match.	Echidna	**Passed**
9	The last updated timestamp on a `PrimitiveEngine` contract never exceeds the maturity of the pool.	Echidna	**Passed**
10	The fee (gamma value) never exceeds 100%.	Echidna	**Passed**

`rmm-core` end-to-end pool creation. The following properties ensure that the system behaves properly when users attempt to create pools.

ID	Property	Tool	Result
11	Creating a pool through `PrimitiveEngine` with the correct preconditions never reverts.	Echidna	**Passed**
12	Creating a pool through `PrimitiveEngine` with an out-of-range gamma always reverts.	Echidna	**Passed**
13	Creating a pool through `PrimitiveEngine` saves a new calibration and liquidity allocation.	Echidna	**Passed**

`rmm-core` end-to-end `deposit` and `withdraw` operations. The following properties ensure the system behaves properly when users deposit and withdraw into the engine.

ID	Property	Tool	Result
14	Depositing into a pool through `PrimitiveEngine` with the correct preconditions never reverts.	Echidna	**Passed**

15	Depositing into a pool with zero risky and zero stable balances always reverts.	Echidna	**Passed**
16	Depositing into a pool always increases the margins for the supplied recipient by the deposited amount.	Echidna	**Passed**
17	Depositing into a pool always decreases the caller's token balance by the deposited amount.	Echidna	**Passed**
18	The margin of the zero address is always zero.	Echidna	TOB-PT V-5
19	Withdrawing from a pool through `PrimitiveEngine` with the correct preconditions always succeeds.	Echidna	**Passed**
20	Withdrawing from a pool with zero risky and zero stable balances always reverts.	Echidna	**Passed**
21	Withdrawing from a pool with the zero address as the recipient always reverts.	Echidna	**Passed**
22	Withdrawing from a pool always results in the decrease of the withdrawn amount in the margins for the supplied recipient.	Echidna	**Passed**
23	Withdrawing from a pool with insufficient margin balances reverts.	Echidna	**Passed**
24	Depositing into a pool and immediately withdrawing from it results in identical margin balances.	Echidna	**Passed**
25	Withdrawing from a pool never results in a change of recipient margin balance.	Echidna	**Passed**
26	Withdrawing from a pool always increases the recipient's token balance by the withdrawn amount.	Echidna	TOB-PT V-14

`rmm-core` end-to-end `allocate` and `remove` operations. The following properties ensure that the system behaves properly when funds are allocated and removed from the system.

ID	Property	Tool	Result
27	Allocating funds to `PrimitiveEngine` with the correct preconditions never reverts.	Echidna	**Passed**
28	Allocating funds from a margin with an insufficient balance to a pool always reverts.	Echidna	**Passed**
29	Allocating funds from a margin to a pool always results in a decrease of the caller's risky and stable margins.	Echidna	**Passed**
30	Allocating funds that do not come from a margin to a pool never changes the margins.	Echidna	**Passed**
31	Allocating funds to a pool always results in an increase in the reserves.	Echidna	**Passed**
32	Allocating funds to an expired pool after maturity always reverts.	Echidna	**Passed**
33	After allocating funds to a pool, calling the inverse of `remove` always succeeds.	Echidna	**Passed**
34	Allocating funds to a pool never results in a decrease of liquidity.	Echidna	**Passed**
35	Allocating funds to a pool always updates the reserve block's timestamp.	Echidna	**Passed**
36	Calling `allocate` and `remove` sequentially with an optimal value amount results in the same amount.	Echidna	TOB-PTV-11
37	Removing funds from a pool with the correct preconditions never reverts.	Echidna	**Passed**
38	Removing funds from a pool that has insufficient liquidity always reverts.	Echidna	**Passed**
39	Removing funds from a pool that has insufficient risky or stable reserves always reverts.	Echidna	**Passed**

40	Removing fewer funds than the total position amount in the engine never fails.	Echidna	**Passed**
41	Removing funds from a pool results in an increase in margins.	Echidna	**Passed**
42	Removing funds never leads to an increase in liquidity.	Echidna	**Passed**

`rmm-core` end-to-end pool swaps. The following properties ensure that the system behaves properly when swaps are being executed.

ID	Property	Tool	Result
43	The timestamp of the pool is updated after a swap.	Echidna	**Passed**
44	The pool invariant always increases.	Echidna	**Passed**
45	After a swap, the reserve balances for tokens accurately reflect the amount paid out.	Echidna	**Passed**

`rmm-manager`. The following properties check behavior in the `rmm-manager` directory. These properties require additional investigation and additional preconditions.

ID	Property	Tool	Result
46	The block timestamp is updated after calls to `allocate`.	Echidna	**WIP**
47	The block timestamp increases between timestamp calls.	Echidna	**WIP**
48	Allocating to reserves always results in an increase of risky and stable amounts.	Echidna	**WIP**
49	Allocating to reserves always results in an increase in liquidity.	Echidna	**WIP**
50	Allocating from a margin through the manager always results in a decrease in margins.	Echidna	**WIP**

51	Allocating funds that do not come from a margin through the manager never changes the margins.	Echidna	**WIP**
52	Allocating funds always results in an increase in ERC1155 tokens of `delLiquidity`.	Echidna	**WIP**
53	Removing funds from reserves always results in a decrease of liquidity.	Echidna	**WIP**
54	Removing funds from reserves results in an update to the timestamp.	Echidna	**WIP**
55	Removing funds from reserves always results in a decrease in `reserveRisky` and `reserveStable`.	Echidna	**WIP**
56	Depositing funds into margins results in an increase in risky and stable margins.	Echidna	**WIP**
57	Depositing zero risky and stable assets reverts.	Echidna	**WIP**

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	Solidity 0.8 with native overflow/underflow support is used throughout the system. The expected behavior and arithmetic of the system is well documented. Using Echidna, we identified certain functions whose inverse checks failed (TOB-PTV-6, TOB-PTV-11, TOB-PTV-12). We highly recommend ensuring that all documentation on parameter fine-tuning is up to date to match the current implementation.	**Moderate**
Auditing	The Primitive codebase has sufficient events for monitoring the system. Primitive also mentioned the use a Discord bot for off-chain monitoring. We recommend creating a thorough incident response plan. Our recommendations for doing so are specified in appendix G.	**Moderate**
Authentication / Access Controls	The privileged actors in the system are limited.	**Satisfactory**
Complexity Management	The functions in the Primitive codebase are small and easy to understand. These functions are also well documented in the white paper and in-code. The use of callbacks in the system may occasionally hinder the ability to test functions in isolation. The lack of an on-chain `getRiskyGivenStable` calculation for swaps can also hinder testing.	**Moderate**
Cryptography and Key Management	No components related to key management or cryptography were in scope for this review.	**Not Applicable**

Decentralization	The Primitive smart contracts are not upgradeable.	**Strong**
Documentation	Primitive provided very detailed documentation through the white paper, the documentation, and code comments. For many of these, the expected behavior matches the implementations, aside from certain out-of-date parameterized fine-tuning.	**Satisfactory**
Front-Running Resistance	We recommend implementing historical mathematical analysis on the pools to identify front-running and arbitrage opportunities.	**Further Investigation Required**
Low-Level Calls	The use of low-level calls in the system is limited. Where low-level calls are required, the consequences of using them are well documented in code comments.	**Satisfactory**
Testing and Verification	The codebase contains adequate unit tests. However, it lacks end-to-end tests for the protocol's integration with the Primitive manager. During the audit, we integrated more advanced testing methods, like fuzzing, to properly test the arithmetic.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Transfer operations may silently fail due to the lack of contract existence checks	Data Validation	High
2	Project dependencies contain vulnerabilities	Patching	Medium
3	Anyone could steal pool tokens' earned interest	Timing	Low
4	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
5	Lack of zero-value checks on functions	Data Validation	Informational
6	uint256.percentage() and int256.percentage() are not inverses of each other	Undefined Behavior	Undetermined
7	Users can allocate tokens to a pool at the moment the pool reaches maturity	Timing	Informational
8	Possible front-running vulnerability during BUFFER time	Timing	Undetermined
9	Inconsistency in allocate and remove functions	Data Validation	Informational
10	Areas of the codebase that would benefit from additional documentation	Data Validation	Informational
11	Allocate and remove are not exact inverses of each other	Undefined Behavior	Medium
12	scaleToX64() and scalefromX64() are not inverses of each other	Undefined Behavior	Undetermined

13	getCDF always returns output in the range of (0, 1)	Undefined Behavior	**Undetermined**
14	Lack of data validation on withdrawal operations	Data Validation	**Medium**

Detailed Findings

1. Transfer operations may silently fail due to the lack of contract existence checks

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PTV-1
Target: `rmm-core/contracts/libraries/Transfers.sol`, `rmm-manager/contracts/libraries/TransferHelper.sol`	

Description

The pool fails to check that a contract exists before performing transfers. As a result, the pool may assume that failed transactions involving destroyed tokens or tokens that have not yet been deployed were successful.

`Transfers.safeTransfer`, `TransferHelper.safeTransfer`, and `TransferHelper.safeTransferFrom` use low-level calls to perform transfers without confirming the contract's existence:

```
) internal {
    (bool success, bytes memory returnData) = address(token).call(
        abi.encodeWithSelector(token.transfer.selector, to, value)
    );
    require(success && (returnData.length == 0 || abi.decode(returnData, (bool))), "Transfer
fail");
}
```

Figure 1.1: `rmm-core/contracts/libraries/Transfers.sol#16-21`

The Solidity documentation includes the following warning:

> *The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.*

Figure 1.2: The Solidity documentation details the necessity of executing existence checks before performing low-level calls.

Therefore, if the tokens to be transferred have not yet been deployed or have been destroyed, `safeTransfer` and `safeTransferFrom` will return success even though the transfer was not executed.

Exploit Scenario

The pool contains two tokens: A and B. The A token has a bug, and the contract is destroyed. Bob is not aware of the issue and swaps 1,000 B tokens for A tokens. Bob successfully transfers 1,000 B tokens to the pool but does not receive any A tokens in return. As a result, Bob loses 1,000 B tokens.

Recommendations

Short term, implement a contract existence check before the low-level calls in `Transfer.safeTransfer`, `TransferHelper.safeTransfer`, and `TransferHelper.safeTransferFrom`. This will ensure that a swap will revert if the token to be bought no longer exists, preventing the pool from accepting the token to be sold without returning any tokens in exchange.

Long term, avoid implementing low-level calls. If such calls are unavoidable, carefully review the Solidity documentation, particularly the "Warnings" section, before implementing them to ensure that they are implemented correctly.

2. Project dependencies contain vulnerabilities

Severity: **Medium**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-PTV-2
Target: `rmm-core`, `rmm-manager`	

Description

Although dependency scans did not indicate a direct threat to the project under review, `yarn audit` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review. The output below details these issues.

CVE ID	Description	Dependency
CVE-2021-32819	Insecure template handling in Squirrelly	`squirrelly`
CVE-2021-23337	Command Injection in `lodash`	`lodash`
CVE-2021-23358	Arbitrary Code Execution in `underscore`	`underscore`

Figure 2.1: Advisories affecting `rmm-core`/`rmm-manager` dependencies

Exploit Scenario

Alice installs the dependencies of an in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure that the code does not use and is not affected by the vulnerable functionality of the dependency.

3. Anyone could steal pool tokens' earned interest

Severity: **Low**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-PTV-3
Target: `rmm-core/contracts/PrimitiveEngine.sol`	

Description

If a `PrimitiveEngine` contract is deployed with certain ERC20 tokens, unexpected token interest behavior could allow token interest to count toward the number of tokens required for the `deposit`, `allocate`, `create`, and swap functions, allowing the user to avoid paying in full.

Liquidity providers use the `deposit` function to increase the liquidity in a position. The following code within the function verifies that the pool has received at least the minimum number of tokens required by the protocol:

```
if (delRisky != 0) balRisky = balanceRisky();
if (delStable != 0) balStable = balanceStable();
IPrimitiveDepositCallback(msg.sender).depositCallback(delRisky, delStable, data); //
agnostic payment
if (delRisky != 0) checkRiskyBalance(balRisky + delRisky);
if (delStable != 0) checkStableBalance(balStable + delStable);
emit Deposit(msg.sender, recipient, delRisky, delStable);
```

Figure 3.1: `rmm-core/contracts/PrimitiveEngine.sol#213-217`

Assume that both `delRisky` and `delStable` are positive. First, the code fetches the current balances of the tokens. Next, the `depositCallback` function is called to transfer the required number of each token to the pool contract. Finally, the code verifies that each token's balance has increased by at least the required amount.

There could be a token that allows token holders to earn interest simply because they are token holders. To retrieve this interest, token holders could call a certain function to calculate the interest earned and increase their balances.

An attacker could call this function from within the `depositCallback` function to pay out interest to the pool contract. This would increase the pool's token balance, decreasing the number of tokens that the user needs to transfer to the pool contract to pass the balance check (i.e., the check confirming that the balance has sufficiently increased). In effect, the

user's token payment obligation is reduced because the interest accounts for part of the required balance increase.

To date, we have not identified a token contract that contains such a functionality; however, it is possible that one exists or could be created.

Exploit Scenario

Bob deploys a `PrimitiveEngine` contract with token1 and token2. Token1 allows its holders to earn passive interest. Anyone can call `get_interest(address)` to make a certain token holder's interest be claimed and added to the token holder's balance. Over time, the pool can claim 1,000 tokens. Eve calls `deposit`, and the pool requires Eve to send 1,000 tokens. Eve calls `get_interest(address)` in the `depositCallback` function instead of sending the tokens, depositing to the pool without paying the minimum required tokens.

Recommendations

Short term, add documentation explaining to users that the use of interest-earning tokens can reduce the standard payments for `deposit`, `allocate`, `create`, and `swap`.

Long term, using the Token Integration Checklist (appendix C), generate a document detailing the shortcomings of tokens with certain features and the impacts of their use in the Primitive protocol. That way, users will not be alarmed if the use of a token with nonstandard features leads to unexpected results.

4. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PTV-4
Target: `rmm-core/hardhat.config.ts`, `rmm-manager/hardhat-config.ts`	

Description
The Primitive 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 Primitive 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.

5. Lack of zero-value checks on functions

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PTV-5
Target: Throughout the code	

Description

Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

```
function deposit(
    address recipient,
    uint256 delRisky,
    uint256 delStable,
    bytes calldata data
) external override lock {
    if (delRisky == 0 && delStable == 0) revert ZeroDeltasError();
    margins[recipient].deposit(delRisky, delStable); // state update

    uint256 balRisky;
    uint256 balStable;
    if (delRisky != 0) balRisky = balanceRisky();
    if (delStable != 0) balStable = balanceStable();
    IPrimitiveDepositCallback(msg.sender).depositCallback(delRisky, delStable, data); //
agnostic payment
    if (delRisky != 0) checkRiskyBalance(balRisky + delRisky);
    if (delStable != 0) checkStableBalance(balStable + delStable);
    emit Deposit(msg.sender, recipient, delRisky, delStable);
}
```

Figure 5.1: rmm-core/contracts/PrimitiveEngine.sol#L201-L219

Among others, the following functions lack zero-value checks on their arguments:

- `PrimitiveEngine.deposit`
- `PrimitiveEngine.withdraw`
- `PrimitiveEngine.allocate`
- `PrimitiveEngine.swap`

- `PositionDescriptor.constructor`
- `MarginManager.deposit`
- `MarginManager.withdraw`
- `SwapManager.swap`
- `CashManager.unwrap`
- `CashManager.sweepToken`

Exploit Scenario
Alice, a user, mistakenly provides the zero address as an argument when depositing for a recipient. As a result, her funds are saved in the margins of the zero address instead of a different address.

Recommendations
Short term, add zero-value checks for all function arguments to ensure that users cannot mistakenly set incorrect values, misconfiguring the system.

Long term, use Slither, which will catch functions that do not have zero-value checks.

6. uint256.percentage() and int256.percentage() are not inverses of each other

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PTV-6
Target: `rmm-core/contracts/libraries.Units.sol`	

Description

The `Units` library provides two `percentage` helper functions to convert unsigned integers to signed `64x64` fixed-point values, and vice versa. Due to rounding errors, these functions are not direct inverses of each other.

```
/// @notice         Converts denormalized percentage integer to a fixed point 64.64 number
/// @dev            Convert unsigned 256-bit integer number into signed 64.64 fixed point
number
/// @param denorm   Unsigned percentage integer with precision of 1e4
/// @return         Signed 64.64 fixed point percentage with precision of 1e4
function percentage(uint256 denorm) internal pure returns (int128) {
    return denorm.divu(PERCENTAGE);
}

/// @notice         Converts signed 64.64 fixed point percentage to a denormalized percetage
integer
/// @param denorm   Signed 64.64 fixed point percentage
/// @return         Unsigned percentage denormalized with precision of 1e4
function percentage(int128 denorm) internal pure returns (uint256) {
    return denorm.mulu(PERCENTAGE);
}
```

Figure 6.1: `rmm-core/contracts/libraries/Units.sol#L53-L66`

These two functions use `ABDKMath64x64.divu()` and `ABDKMath64x64.mulu()`, which both round *downward* toward zero. As a result, if a `uint256` value is converted to a signed `64x64` fixed point and then converted back to a `uint256` value, the result will not equal the original `uint256` value:

```
function scalePercentages(uint256 value) public {
        require(value > Units.PERCENTAGE);
        int128 signedPercentage =  value.percentage();
        uint256 unsignedPercentage = signedPercentage.percentage();
```

```
        if(unsignedPercentage != value) {
                emit AssertionFailed("scalePercentages", signedPercentage,
unsignedPercentage);
                assert(false);
        }
```

Figure 6.2: rmm-core/contracts/LibraryMathEchidna.sol#L48-L57

Trail of Bits used Echidna to determine this property violation:

```
Analyzing contract: /rmm-core/contracts/LibraryMathEchidna.sol:LibraryMathEchidna
scalePercentages(uint256): failed!💥
  Call sequence:
    scalePercentages(10006)

Event sequence: Panic(1), AssertionFailed("scalePercentages", 18457812120153777346, 10005)
```

Figure 6.3: Echidna results

Exploit Scenario

1. `uint256.percentage()` – `10006.percentage()` = `1.0006`, which truncates down to 1.
2. `int128.percentage()` – `1.percentage()` = `10000`.
3. The assertion fails because `10006 != 10000`.

Recommendations

Short term, either remove the `int128.percentage()` function if it is unused in the system or ensure that the percentages round in the correct direction to minimize rounding errors.

Long term, use Echidna to test system and mathematical invariants.

7. Users can allocate tokens to a pool at the moment the pool reaches maturity

Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-PTV-7
Target: `rmm-core/contracts/PrimitiveEngine.sol`	

Description

Users can allocate tokens to a pool at the moment the pool reaches maturity, which creates an opportunity for attackers to front-run or update the curve right before the maturity period ends.

```
function allocate(
    bytes32 poolId,
    address recipient,
    uint256 delRisky,
    uint256 delStable,
    bool fromMargin,
    bytes calldata data
) external override lock returns (uint256 delLiquidity) {
    if (delRisky == 0 || delStable == 0) revert ZeroDeltasError();
    Reserve.Data storage reserve = reserves[poolId];
    if (reserve.blockTimestamp == 0) revert UninitializedError();
    uint32 timestamp = _blockTimestamp();
    if (timestamp > calibrations[poolId].maturity) revert PoolExpiredError();

    uint256 liquidity0 = (delRisky * reserve.liquidity) / uint256(reserve.reserveRisky);
    uint256 liquidity1 = (delStable * reserve.liquidity) / uint256(reserve.reserveStable);
    delLiquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
    if (delLiquidity == 0) revert ZeroLiquidityError();

    liquidity[recipient][poolId] += delLiquidity; // increase position liquidity
    reserve.allocate(delRisky, delStable, delLiquidity, timestamp); // increase reserves and
liquidity

    if (fromMargin) {
        margins.withdraw(delRisky, delStable); // removes tokens from `msg.sender` margin
account
    } else {
        (uint256 balRisky, uint256 balStable) = (balanceRisky(), balanceStable());
        IPrimitiveLiquidityCallback(msg.sender).allocateCallback(delRisky, delStable, data);
// agnostic payment
```

```
        checkRiskyBalance(balRisky + delRisky);
        checkStableBalance(balStable + delStable);
    }

    emit Allocate(msg.sender, recipient, poolId, delRisky, delStable);
}
```

Figure 7.1: rmm-core/contracts/PrimitiveEngine.sol#L236-L268

Recommendations

Short term, document the expected behavior of transactions to allocate funds into a pool that has just reached maturity and analyze the front-running risk.

Long term, analyze all front-running risks on all transactions in the system.

8. Possible front-running vulnerability during BUFFER time

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Timing	Finding ID: TOB-PTV-8
Target: `rmm-core/contracts/PrimitiveEngine.sol`	

Description

The `PrimitiveEngine.swap` function permits swap transactions until 120 seconds after maturity, which could enable miners to front-run swap transactions and engage in malicious behavior. The constant tau value may allow miners to profit from front-running transactions when the swap curve is locked after maturity.

```
SwapDetails memory details = SwapDetails({
    recipient: recipient,
    poolId: poolId,
    deltaIn: deltaIn,
    deltaOut: deltaOut,
    riskyForStable: riskyForStable,
    fromMargin: fromMargin,
    toMargin: toMargin,
    timestamp: _blockTimestamp()
});

uint32 lastTimestamp = _updateLastTimestamp(details.poolId); // updates lastTimestamp of
`poolId`
if (details.timestamp > lastTimestamp + BUFFER) revert PoolExpiredError(); // 120s buffer to
allow final swaps
```

Figure 8.1: rmm-core/contracts/PrimitiveEngine.sol#L314-L326

Recommendations

Short term, perform an off-chain analysis on the curve and the swaps to determine the impact of a front-running attack on these transactions.

Long term, perform an additional economic analysis with historical data on pools to determine the impact of front-running attacks on all functionality in the system.

9. Inconsistency in allocate and remove functions

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PTV-9
Target: `rmm-core/contracts/PrimitiveEngine.sol`	

Description

The `allocate` and `remove` functions do not have the same interface, as one would expect. The `allocate` function allows users to set the recipient of the allocated liquidity and choose whether the funds will be taken from the `margins` or sent directly. The `remove` function unallocates the liquidity from the pool and sends the tokens to the `msg.sender`; with this function, users cannot set the recipient of the tokens or choose whether the tokens will be credited to their `margins` for future use or directly sent back to them.

```
function allocate(
    bytes32 poolId,
    address recipient,
    uint256 delRisky,
    uint256 delStable,
    bool fromMargin,
    bytes calldata data
) external override lock returns (uint256 delLiquidity) {
    if (delRisky == 0 || delStable == 0) revert ZeroDeltasError();
    Reserve.Data storage reserve = reserves[poolId];
    if (reserve.blockTimestamp == 0) revert UninitializedError();
    uint32 timestamp = _blockTimestamp();
    if (timestamp > calibrations[poolId].maturity) revert PoolExpiredError();

    uint256 liquidity0 = (delRisky * reserve.liquidity) / uint256(reserve.reserveRisky);
    uint256 liquidity1 = (delStable * reserve.liquidity) / uint256(reserve.reserveStable);
    delLiquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
    if (delLiquidity == 0) revert ZeroLiquidityError();

    liquidity[recipient][poolId] += delLiquidity; // increase position liquidity
    reserve.allocate(delRisky, delStable, delLiquidity, timestamp); // increase reserves and
liquidity

    if (fromMargin) {
        margins.withdraw(delRisky, delStable); // removes tokens from `msg.sender` margin
account
    } else {
```

```
        (uint256 balRisky, uint256 balStable) = (balanceRisky(), balanceStable());
        IPrimitiveLiquidityCallback(msg.sender).allocateCallback(delRisky, delStable, data);
// agnostic payment
        checkRiskyBalance(balRisky + delRisky);
        checkStableBalance(balStable + delStable);
    }

    emit Allocate(msg.sender, recipient, poolId, delRisky, delStable);
}
```

Figure 9.1: rmm-core/contracts/PrimitiveEngine.sol#L236-L268

Recommendations

Short term, either document the design decision or add the logic to the `remove` function allowing users to set the recipient and to choose whether the tokens should be credited to their `margins`.

Long term, make sure to document design decisions and the rationale behind them, especially for behavior that may not be obvious.

10. Areas of the codebase that are inconsistent with the documentation

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PTV-10
Target: `rmm-core/contracts/PrimitiveEngine.sol`	

Description
The Primitive codebase contains clear documentation and mathematical analysis denoting the intended behavior of the system. However, we identified certain areas in which the implementation does not match the white paper, including the following:

- **Expected range for the gamma value of a pool.** The white paper defines 10,000 as 100% in the smart contract; however, the contract checks that the provided gamma is between 9,000 (inclusive) and 10,000 (exclusive); if it is not within this range, the pool reverts with a `GammaError`.

The white paper should be updated to reflect the behavior of the code in these areas.

Recommendations
Short term, review and properly document all areas of the codebase with this gamma range check.

Long term, ensure that the formal specification matches the expected behavior of the protocol.

11. Allocate and remove are not exact inverses of each other

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PTV-11
Target: `rmm-core/contracts/libraries/Reserve.sol`	

Description

Due to the rounding logic used in the codebase, when users allocate funds into a system, they may not receive the same amount back when they remove them.

When funds are allocated into a system, the values are rounded down (through native truncation) when they are added to the reserves:

```
/// @notice                 Add to both reserves and total supply of liquidity
/// @param  reserve         Reserve storage to manipulate
/// @param  delRisky        Amount of risky tokens to add to the reserve
/// @param  delStable       Amount of stable tokens to add to the reserve
/// @param  delLiquidity    Amount of liquidity created with the provided tokens
/// @param  blockTimestamp  Timestamp used to update cumulative reserves
function allocate(
    Data storage reserve,
    uint256 delRisky,
    uint256 delStable,
    uint256 delLiquidity,
    uint32 blockTimestamp
) internal {
    update(reserve, blockTimestamp);
    reserve.reserveRisky += delRisky.toUint128();
    reserve.reserveStable += delStable.toUint128();
    reserve.liquidity += delLiquidity.toUint128();
}
```

Figure 11.1: `rmm-core/contracts/libraries/Reserve.sol#L70-L87`

When funds are removed from the reserves, they are similarly truncated:

```
/// @notice                 Remove from both reserves and total supply of liquidity
/// @param  reserve         Reserve storage to manipulate
/// @param  delRisky        Amount of risky tokens to remove to the reserve
/// @param  delStable       Amount of stable tokens to remove to the reserve
/// @param  delLiquidity    Amount of liquidity removed from total supply
/// @param  blockTimestamp  Timestamp used to update cumulative reserves
```

```
function remove(
    Data storage reserve,
    uint256 delRisky,
    uint256 delStable,
    uint256 delLiquidity,
    uint32 blockTimestamp
) internal {
    update(reserve, blockTimestamp);
    reserve.reserveRisky -= delRisky.toUint128();
    reserve.reserveStable -= delStable.toUint128();
    reserve.liquidity -= delLiquidity.toUint128();
}
```

Figure 11.2: rmm-core/contracts/libraries/Reserve.sol#L89-L106

We used the following Echidna property to test this behavior:

```
function check_allocate_remove_inverses(
    uint256 randomId,
    uint256 intendedLiquidity,
    bool fromMargin
) public {
    AllocateCall memory allocate;
    allocate.poolId = Addresses.retrieve_created_pool(randomId);
    retrieve_current_pool_data(allocate.poolId, true);
    intendedLiquidity = E2E_Helper.one_to_max_uint64(intendedLiquidity);
    allocate.delRisky = (intendedLiquidity * precall.reserve.reserveRisky) /
precall.reserve.liquidity;
    allocate.delStable = (intendedLiquidity * precall.reserve.reserveStable) /
precall.reserve.liquidity;

    uint256 delLiquidity = allocate_helper(allocate);

    // these are calculated the amount returned when remove is called
    (uint256 removeRisky, uint256 removeStable) = remove_should_succeed(allocate.poolId,
delLiquidity);
    emit AllocateRemoveDifference(allocate.delRisky, removeRisky);
    emit AllocateRemoveDifference(allocate.delStable, removeStable);

    assert(allocate.delRisky == removeRisky);
    assert(allocate.delStable == removeStable);
    assert(intendedLiquidity == delLiquidity);
}
```

Figure 11.3: rmm-core/contracts/libraries/Reserve.sol#L89-L106

In considering this rounding logic, we used Echidna to calculate the most optimal `allocate` value for an amount of liquidity, which resulted 1,920,041,647,503 as the difference in the amount allocated and the amount removed.

```
check_allocate_remove_inverses(uint256,uint256,bool): failed!💥
  Call sequence:

create_new_pool_should_not_revert(113263940847354084267525170308314,0,12,58,414705177,292070
354338709387317704910944590379491006113120533898160371690233992451 74) from:
0x0000000000000000000000000000000000020000 Gas: 0xbebc20

check_allocate_remove_inverses(513288669432172152578276403318402760987129411133329015270396,
675391606931488162786753316903883654910567233327356334685,false) from:
0x1E2F9E10D02a6b8F8f69fcBf515e75039D2EA30d

Event sequence: Panic(1), Transfer(6361150874), Transfer(6430226091720657429 4870),
AllocateMarginBalance(0, 0, 6361150874, 6430226091720657429 4870), Transfer(6361150874),
Transfer(6430226091720657429 4870), Allocate(6361150874, 6430226091720657429 4870),
Remove(6361150873, 6430226091528653264 7367), AllocateRemoveDifference(6361150874,
6361150873), AllocateRemoveDifference(6430226091720657429 4870, 6430226091528653264 7367)
```

Figure 11.4: Echidna results

Exploit Scenario

Alice, a Primitive user, determines a specific amount of liquidity that she wants to put into the system. She calculates the required risky and stable tokens to make the trade, and then allocates the funds to the pool. Due to the rounding direction in the `allocate` operation and the pool, she receives less than she expected after removing her liquidity.

Recommendations

Short term, perform additional analysis to determine a safe delta value to allow the `allocate` and `remove` operations to happen. Document this issue for end users to ensure that they are aware of the rounding behavior.

Long term, use Echidna to test system and mathematical invariants.

12. scaleToX64() and scalefromX64() are not inverses of each other

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PTV-12
Target: `rmm-core/contracts/libraries/Units.sol`	

Description

The `Units` library provides the `scaleToX64()` and `scalefromX64()` helper functions to convert unsigned integers to signed `64x64` fixed-point values, and vice versa. Due to rounding errors, these functions are not direct inverses of each other.

```
/// @notice             Converts unsigned 256-bit wei value into a fixed point 64.64 number
/// @param   value      Unsigned 256-bit wei amount, in native precision
/// @param   factor     Scaling factor for `value`, used to calculate decimals of `value`
/// @return  y          Signed 64.64 fixed point number scaled from native precision
function scaleToX64(uint256 value, uint256 factor) internal pure returns (int128 y) {
    uint256 scaleFactor = PRECISION / factor;
    y = value.divu(scaleFactor);
}
```

Figure 12.1: rmm-core/contracts/libraries/Units.sol#L35-L42

These two functions use `ABDKMath64x64.divu()` and `ABDKMath64x64.mulu()`, which both round *downward* toward zero. As a result, if a `uint256` value is converted to a signed `64x64` fixed point and then converted back to a `uint256` value, the result will not equal the original `uint256` value:

```
/// @notice             Converts signed fixed point 64.64 number into unsigned 256-bit wei
value
/// @param   value      Signed fixed point 64.64 number to convert from precision of 10^18
/// @param   factor     Scaling factor for `value`, used to calculate decimals of `value`
/// @return  y          Unsigned 256-bit wei amount scaled to native precision of 10^(18 -
factor)
function scalefromX64(int128 value, uint256 factor) internal pure returns (uint256 y) {
    uint256 scaleFactor = PRECISION / factor;
    y = value.mulu(scaleFactor);
}
```

Figure 12.2: rmm-core/contracts/libraries/Units.sol#L44-L51

We used the following Echidna property to test this behavior:

```
function scaleToAndFromX64Inverses(uint256 value, uint256 _decimals) public {
        // will enforce factor between 0 - 12
        uint256 factor = _decimals % (13);
        // will enforce scaledFactor between 1 - 10**12 , because 10**0 = 1
        uint256 scaledFactor = 10**factor;

        int128 scaledUpValue = value.scaleToX64(scaledFactor);
        uint256 scaledDownValue = scaledUpValue.scalefromX64(scaledFactor);

        assert(scaledDownValue == value);
}
```

Figure 12.3: `contracts/crytic/LibraryMathEchidna.sol`

```
scaleToAndFromX64Inverses(uint256,uint256): failed!💥
  Call sequence:
    scaleToAndFromX64Inverses(1,0)

Event sequence: Panic(1)
```

Figure 12.4: Echidna results

Recommendations

Short term, ensure that the percentages round in the correct direction to minimize rounding errors.

Long term, use Echidna to test system and mathematical invariants.

13. getCDF always returns output in the range of (0, 1)

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PTV-13
Target: `rmm-core/contracts/libraries/CumulativeNormalDistribution.sol`	

Description

`CumulativeNormalDistribution` provides the `getCDF` function to calculate an approximation of the cumulative distribution function, which should result in `(0, 1]`; however, the `getCDF` function could return 1.

```
/// @notice Uses Abramowitz and Stegun approximation:
///         https://en.wikipedia.org/wiki/Abramowitz_and_Stegun
/// @dev    Maximum error: 3.15x10-3
/// @return Standard Normal Cumulative Distribution Function of `x`
function getCDF(int128 x) internal pure returns (int128) {
    int128 z = x.div(CDF3);
    int128 t = ONE_INT.div(ONE_INT.add(CDF0.mul(z.abs())));
    int128 erf = getErrorFunction(z, t);
    if (z < 0) {
        erf = erf.neg();
    }
    int128 result = (HALF_INT).mul(ONE_INT.add(erf));
    return result;
}
```

Figure 13.1: rmm-core/contracts/libraries/CumulativeNormalDistribution.sol#L24-L37

We used the following Echidna property to test this behavior.

```
function CDFCheckRange(uint128 x, uint128 neg) public {
        int128 x_x = realisticCDFInput(x, neg);

        int128 res = x_x.getCDF();
        emit P(x_x, res, res.toInt());
        assert(res > 0 && res.toInt() < 1);
}
```

Figure 13.2: rmm-core/contracts/LibraryMathEchidna.sol

```
CDFCheckRange(uint128,uint128): failed!💥
```

```
  Call sequence:
    CDFCheckRange(16895162281582749303 7,148697375557466323561959026665 1)

Event sequence: Panic(1), P(16895162281582749303 7, 18446744073709551616, 1)
```

Figure 13.3: Echidna results

Recommendations

Short term, perform additional analysis to determine whether this behavior is an issue for the system.

Long term, use Echidna to test system and mathematical invariants.

14. Lack of data validation on withdrawal operations

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PTV-13
Target: `rmm-core/contracts/PrimitiveEngine.sol`	

Description

The `withdraw` function allows users to specify the recipient to send funds to. Due to a lack of data validation, the address of the engine could be set as the recipient. As a result, the tokens will be transferred directly to the engine itself.

```
/// @inheritdoc IPrimitiveEngineActions
function withdraw(
    address recipient,
    uint256 delRisky,
    uint256 delStable
) external override lock {
    if (delRisky == 0 && delStable == 0) revert ZeroDeltasError();
    margins.withdraw(delRisky, delStable); // state update
    if (delRisky != 0) IERC20(risky).safeTransfer(recipient, delRisky);
    if (delStable != 0) IERC20(stable).safeTransfer(recipient, delStable);
    emit Withdraw(msg.sender, recipient, delRisky, delStable);
}
```

Figure 14.1: rmm-core/contracts/PrimitiveEngine.sol#L221-L232

We used the following Echidna property to test this behavior.

```
function withdraw_with_only_non_zero_addr(
    address recipient,
    uint256 delRisky,
    uint256 delStable
) public {
    require(recipient != address(0));
    //ensures that delRisky and delStable are at least 1 and not too large to overflow the
deposit
    delRisky = E2E_Helper.one_to_max_uint64(delRisky);
    delStable = E2E_Helper.one_to_max_uint64(delStable);
    MarginHelper memory senderMargins = populate_margin_helper(address(this));
    if (senderMargins.marginRisky < delRisky || senderMargins.marginStable < delStable) {
        withdraw_should_revert(recipient, delRisky, delStable);
    } else {
```

```
        withdraw_should_succeed(recipient, delRisky, delStable);
    }
}
function withdraw_should_succeed(
    address recipient,
    uint256 delRisky,
    uint256 delStable
) internal {
    MarginHelper memory precallSender = populate_margin_helper(address(this));
    MarginHelper memory precallRecipient = populate_margin_helper(recipient);
    uint256 balanceRecipientRiskyBefore = risky.balanceOf(recipient);
    uint256 balanceRecipientStableBefore = stable.balanceOf(recipient);
    uint256 balanceEngineRiskyBefore = risky.balanceOf(address(engine));
    uint256 balanceEngineStableBefore = stable.balanceOf(address(engine));

    (bool success, ) = address(engine).call(
        abi.encodeWithSignature("withdraw(address,uint256,uint256)", recipient, delRisky,
delStable)
    );
    if (!success) {
        assert(false);
        return;
    }

    {
        assert_post_withdrawal(precallSender, precallRecipient, recipient, delRisky,
delStable);
        //check token balances
        uint256 balanceRecipientRiskyAfter = risky.balanceOf(recipient);
        uint256 balanceRecipientStableAfter = stable.balanceOf(recipient);
        uint256 balanceEngineRiskyAfter = risky.balanceOf(address(engine));
        uint256 balanceEngineStableAfter = stable.balanceOf(address(engine));
        emit DepositWithdraw("balance recip risky", balanceRecipientRiskyBefore,
balanceRecipientRiskyAfter, delRisky);
        emit DepositWithdraw("balance recip stable", balanceRecipientStableBefore,
balanceRecipientStableAfter, delStable);
        emit DepositWithdraw("balance engine risky", balanceEngineRiskyBefore,
balanceEngineRiskyAfter, delRisky);
        emit DepositWithdraw("balance engine stable", balanceEngineStableBefore,
balanceEngineStableAfter, delStable);
        assert(balanceRecipientRiskyAfter == balanceRecipientRiskyBefore + delRisky);
        assert(balanceRecipientStableAfter == balanceRecipientStableBefore + delStable);
        assert(balanceEngineRiskyAfter == balanceEngineRiskyBefore - delRisky);
        assert(balanceEngineStableAfter == balanceEngineStableBefore - delStable);
    }
}
```

Figure 14.2: rmm-core/contracts/crytic/E2E_Deposit_Withdrawal.sol

```
withdraw_with_safe_range(address,uint256,uint256): failed!💥
  Call sequence:

deposit_with_safe_range(0xa329c0648769a73afac7f9381e08fb43dbea72,1157920892373161954235709850086879078532699846656405640394475840079131296399375964323976539599410180707317759394870432162568223259259646265020558109612095 5) from: 0x1E2F9E10D02a6b8F8f69fcBf515e75039D2EA30d

withdraw_with_safe_range(0x48bacb9266a570d521063ef5dd96e61686dbe788,5248038478797710845,748) from: 0x6A4A62E5A7eD13c361b176A5F62C2eE620Ac0DF8

Event sequence: Panic(1), Transfer(5248038478797710846), Transfer(749),
Withdraw(5248038478797710846, 749), DepositWithdraw("sender risky", 8446744073709551632,
3198705594911840786, 5248038478797710846), DepositWithdraw("sender stable",
15594018607531992466, 15594018607531991717, 749), DepositWithdraw("balance recip risky",
8446744073709551632, 8446744073709551632, 5248038478797710846), DepositWithdraw("balance
recip stable", 15594018607531992466, 15594018607531992466, 749), DepositWithdraw("balance
engine risky", 8446744073709551632, 8446744073709551632, 5248038478797710846),
DepositWithdraw("balance engine stable", 15594018607531992466, 15594018607531992466, 749)
```

Figure 14.3: Echidna results

Exploit Scenario

Alice, a user, withdraws her funds from the Primitive engine. She accidentally specifies the address of the recipient as the engine address, and her funds are left stuck in the contract.

Recommendations

Short term, add a check to ensure that users cannot withdraw to the engine address directly to ensure that users are protected from these mistakes.

Long term, use Echidna to test system and mathematical invariants.

Summary of Recommendations

The Primitive codebase is a work in progress with multiple planned iterations. Trail of Bits recommends that Primitive address the findings detailed in this report and take the following additional steps prior to deployment:

- Integrate Slither into a continuous integration pipeline to detect common issues.
- Continue extending the fuzzing tests to ensure the correctness of the arithmetic operations. In all cases in which rounding is used, ensure that the code always rounds in a direction that is favorable to the pool. Our recommendations for additional properties are highlighted in appendix F.
- Develop a detailed incident response plan to ensure that any issues that arise can be addressed promptly and without confusion.

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 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	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. Token Integration Checklist

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

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. Fixed-Point Rounding Recommendations

Primitive uses fixed-point arithmetic throughout the system. This strategy requires that numbers round up or down, which may lead to dust that is beneficial for an attacker. While we are still investigating the assumptions of these deviations, we offer some recommendations on the expected rounding direction of these values.

Determining Rounding Direction

To determine how to apply rounding (whether up or down), consider the result of the expected output.

For example, the formula for a swap of token x for token y calculates how much of token x must be sent to the contract to receive y.

$$y' = K\phi(\phi^{-1}(1 - x + \gamma\Delta)) - \sigma\sqrt{\tau}) + k$$

In order to benefit the pool, y ' must tend toward a lower value (↘) to minimize the amount paid out. As a result, the following should hold:

- $K\phi(\phi^{-1}(1 - x + \gamma\Delta))$ must round ↘
- $\sigma\sqrt{\tau}$ must round ↗
  - $\sigma\nearrow\sqrt{\tau\nearrow}$
- k must round ↘

Therefore, the mathematics in the formula should perform this check:

$$y'\searrow = K\searrow\phi\searrow(\phi^{-1}(1 - x + \gamma\Delta)\searrow) - \sigma\nearrow\sqrt{\tau\nearrow}) + k\searrow$$

Similar rounding techniques can be applied in all the system's formulas to ensure that rounding always occurs in the direction that benefits Primitive.

(1-x) Rounding

Several operations require the system to compute $(1 - x)$. The following describes the rules to apply the rounding:

- $(1 - x)\nearrow$ requires $x\searrow$
- $(1 - x)\searrow$ requires $x\nearrow$

E. 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.

- **Replace `variable == false` with `!variable`.** This will improve code readability.

```
   if (EngineAddress.isContract(engine) == false)
```

Figure D.1: rmm-managers/contracts/PrimitiveManager.sol#L53

```
   if (EngineAddress.isContract(engine) == false)
```

Figure D.2: rmm-managers/contracts/PrimitiveManager.sol#L89

```
   if (EngineAddress.isContract(engine) == false)
```

Figure D.3: rmm-managers/contracts/base/MarginManager.sol#L34

```
   if (EngineAddress.isContract(engine) == false)
```

Figure D.4: rmm-managers/contracts/base/SwapManager.sol#L37

```
   if (success == false)
```

Figure D.5: rmm-managers/contracts/libraries/TransferHelper.sol#L68

- **Replace `memory params`'s data location with `calldata`.** This will improve gas usage.

```
   function swap(SwapParams memory params) external payable override lock
checkDeadline(params.deadline) {
```

Figure D.6: rmm-managers/contracts/base/SwapManager.sol#L29

- **Remove unused parameters.** This will improve code readability and will make the codebase easier to maintain, modify, and audit.

```
   function render(address engine, uint256 tokenId) external pure override returns
(string memory) {
        return
            string(
                abi.encodePacked(
                    "",
                    Base64.encode(
                        bytes(
                            ...
                        )
                    )
```

```
                )
            );
    }
```

Figure D.7: rmm-managers/contracts/base/SwapManager.sol#L10-L24

- **Use `bytes constant private _empty = "";`.** This will improve gas usage and code readability, showing users that the variable is constant.

```
    /// @dev  Empty variable to pass to the _mint function
    bytes private _empty;
```

Figure D.8: rmm-managers/contracts/base/PositionManager.sol#L16-L17

- **Use the same `lock` modifier in `rmm-manager` and `rmm-core`.** This will improve gas usage and code readability by using the same `lock` implementation.

```
modifier lock() {
     if (_unlocked != 1) revert LockedError();

     _unlocked = 0;
     _;
     _unlocked = 1;
 }
```

Figure D.9: rmm-managers/contracts/base/Reentrancy.sol#L14-L20

```
modifier lock() {
     if (locked != 1) revert LockedError();

     locked = 2;
     _;
     locked = 1;
 }
```

Figure D.10: rmm-core/contracts/PrimitiveEngine.sol#L75-L81

- **Use consistent language across the codebase.**
    - Rename `scalefromX64` to `scaleFromX64`.

```
  function scalefromX64(int128 value, uint256 factor) internal pure returns (uint256 y)
```

Figure D.11: rmm-core/contracts/libraries/Units.sol#L48

    - Rename `percentage(uint256)` to `percentageToX64`.

```
  function percentage(uint256 denorm) internal pure returns (int128)
```

Figure D.12: rmm-core/contracts/libraries/Units.sol#L57

    - Rename `percentage(int128)` to `percentageFromX64`.

```
function percentage(int128 denorm) internal pure returns (uint256)
```

Figure D.13: rmm-core/contracts/libraries/Units.sol#L64

F. Additional Recommended Properties

In this section, we recommend additional extensions of Echidna to consider after the audit.

General Properties

- **Identify the safe deltas for differences in system values.** These deltas will also help with refining Echidna properties and checking preconditions.

Libraries

- **Check that the paper reimplementation and the library implementation revert accordingly.** This will allow Echidna to compare reverting cases of the CDF functions.

rmm-core

- **Extend the Echidna tests to cover a large input safe bound.** Currently, the Echidna inputs are limited to `[1, uint64.max]` inclusively. We recommend extending this limit to explore additional system states to check that expected behavior surrounding larger numbers is correct.
- **Implement monotonically increasing pool invariant checks.** This will allow Echidna to check that the invariant of the system is always increasing.
- **Extend Echidna to test against multiple engine configurations with different token decimals and scale factors.** The current Echidna configuration points to an engine with two tokens of 18 decimal points. Extending this would allow Echidna to explore additional states.

rmm-manager

- **Add additional precondition checks on `allocate` and `remove`.** Due to time constraints, the `PrimitiveManager` tests are likely missing additional preconditions checks.
- **Integrate tests of `deposit`, `withdraw`, and `swap` in `PrimitiveManager`.**
- **Implement checks to verify that the margins in the manager always match the margin balances in the engine.**

G. Incident Response Recommendations

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

- **Identify who (either specific people or roles) is responsible for carrying out the mitigations (deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
    - Specifying these roles will strengthen the incident response plan and ease the execution of mitigating actions when necessary.
- **Document internal processes for situations in which a deployed remediation does not work or introduces a new bug.**
    - Consider adding a fallback scenario that describes an action plan in the event of a failed remediation.
- **Clearly describe the intended process of contract deployment.**
- **Consider whether and under what circumstances Primitive will make affected users whole after certain issues occur.**
    - Some scenarios to consider include an individual or aggregate loss, a loss resulting from user error, a contract flaw, and a third-party contract flaw.
- **Document how Primitive plans keep up to date on new issues, both to inform future development and to secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - For each language and component, describe the noteworthy sources for vulnerability news. Subscribe to updates for each source. Consider creating a special private Discord channel with a bot that will post the latest vulnerability news; this will help the team keep track of updates all in one place. Also consider assigning specific team members to keep track of the vulnerability news of a specific component of the system.
- **Consider scenarios involving issues that would indirectly affect the system.**
- **Determine when and how the team would reach out to and onboard external parties (auditors, affected users, other protocol developers, etc.).**
    - Some issues may require collaboration with external parties to efficiently remediate them.

- **Define contract behavior that is considered abnormal for off-chain monitoring.**
    - Consider adding more resilient solutions for detection and mitigation, especially in terms of specific alternate endpoints and queries for different data as well as status pages and support contacts for affected services.
- **Combine issues and determine whether new detection and mitigation scenarios are needed.**
- **Perform periodic dry runs of specific scenarios in the incident response plan to find gaps and opportunities for improvement and to develop muscle memory.**
    - Document the intervals at which the team should perform dry runs of the various scenarios. For scenarios that are more likely to happen, perform dry runs more regularly. Create a template to be filled in after a dry run to describe the improvements that need to be made to the incident response.

F. Fix Log

On February 11, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Primitive team for the issues identified in this report. The Primitive team fixed three of the issues reported in the original assessment, partially fixed one, and acknowledged but did not fix the other 10. 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	Transfer operations may silently fail due to the lack of contract existence checks	High	Not Fixed
2	Project dependencies contain vulnerabilities	Medium	Partially Fixed
3	Anyone could steal pool tokens' earned interest	Low	Not Fixed
4	Solidity compiler optimizations can be problematic	Informational	Not Fixed
5	Lack of zero-value checks on functions	Informational	Not Fixed
6	uint256.percentage() and int256.percentage() are not inverses of each other	Undetermined	Fixed
7	Users can allocate tokens to a pool at the moment the pool reaches maturity	Informational	Fixed
8	Possible front-running vulnerability during BUFFER time	Undetermined	Not Fixed
9	Inconsistency in allocate and remove functions	Informational	Not Fixed
10	Areas of the codebase that are inconsistent with the documentation	Informational	Fixed

11	Allocate and remove are not exact inverses of each other	Medium	Not Fixed
12	scaleToX64() and scalefromX64() are not inverses of each other	Undetermined	Not Fixed
13	getCDF always returns output in the range of (0, 1)	Undetermined	Not Fixed
14	Lack of data validation on withdrawal operations	Medium	Not Fixed

Detailed Fix Log

TOB-PTV-2: Project dependencies contain vulnerabilities
Partially fixed. The `underscore`, `lodash`, and `follow-redirects` packages were updated in the core folder (`2d5ace5`) and the manager directory (`f79ed0f`). However, `yarn audit` indicates that additional vulnerable dependencies remain in the codebase with `node-fetch`.

Primitive stated the following:

> *The vulnerable packages will be updated once safe versions are made available*.

TOB-PTV-6: uint256.percentage() and int256.percentage() are not inverses of each other
Fixed. The `percentage(int128 denorm)` function was removed from the codebase (`d35e4c0`).

TOB-PTV-7: Users can allocate tokens to a pool at the moment the pool reaches maturity
Fixed. Primitive changed the allocation requirements to "allow allocation to happen post-maturity indefinitely." (`ad00bcb`)

TOB-PTV-9: Inconsistency in allocate and remove functions
Not fixed. Primitive added documentation explaining the differences between the values passed to these functions (Integration Checklist).

TOB-PTV-10: Areas of the codebase that are inconsistent with the documentation
Fixed. Primitive updated the maximum gamma bound to match the specifications in the white paper (`1b625a7`).