PUBLIC

Code Assessment

of the Hyper RMM & RMM utils

Smart Contracts

June 14, 2022

Produced for

PRIMITIVE

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Alexander and Clément,

Thank you for trusting us to help Primitive Finance with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Hyper RMM & RMM utils according to Scope to support you in forming an opinion on their security risks.

The report and executive summary is considered for internal use only and includes wording emphasising frankness over holisticness to highlight current issues. An updated version of this report, especially one targeting public release, will reflect the security of the future system only and will be written with the outside reader in mind to give a representative impression of the overall security.

The review covered 4 repositories:

1. `Hyper` - repository with an implementation of Primitive RMM, that is optimised for EVM chains where off-chain information is expensive relative to on-chain processing.
2. `rmm-examples` - repository that showcases how contracts can interact with the Primitive V2 RMM contracts.
3. `bits` - repository with gas optimised `ERC20` and `ERC1155` implementations.
4. `migrator` - repository with `Migrator` contract that allows easy migration of Uniswap V2 and V3 assets to the Primitive V2 RMM.

The most critical subjects covered in our audit are asset solvency (ensuring the contract can repay its debt), functional correctness and arithmetic operations correctness. On the reviewed code base security regarding the asset solvency is low as we found two critical issues, Compiler Internal User Balances Not Accounted for in Settlement and LiquidityManager Mixes the Liquidity From Different Pools. Functional correctness is low due to multiple issues like ERC1155 Incorrect Return Value in supportsInterface and ERC1155 Incorrect Function Selectors. Arithmetic operations correctness is low due to Function HyperSwap.getInvariant Can Fail Due to Overflow and HyperLiquidity Unsafe Cast of IDs.

The general subjects covered are gas efficiency, specification and documentation. Gas efficiency is improvable due to some inefficiencies, see Gas Optimisations. Specification is improvable due to Hyper decoder behavior, see Decoder No Length Check and Hyper Canonical Form of Encoding Not Enforced. Documentation is improvable due to mismatches between code and docs, see Incorrect Encoding Documentation and Instructions Incorrect NatSpec.

In summary, we find that the codebase currently contains a high amount of issues which causes the overall codebase to provide low security. At the current state, the code should not be deployed and used in production. Besides fixing the issues raised in this report, we suggest that you expand the test suite to cover more functionality and corner cases. While projects with a high number of issues or very critical issues can require a re-audit, for Hyper RMM & RMM utils, a focused iteration to review fixes for the issues in this report is likely to result in an overall codebase with high security. In case new features are added or larger redesigns or refactorings are conducted, the time reserved for validating the fixes won't be sufficient and it is important to discuss timelines and scope of another audit round as soon as possible.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	2
• Acknowledged	2
High-Severity Findings	7
• Acknowledged	7
Medium-Severity Findings	10
• Acknowledged	10
Low-Severity Findings	14
• Acknowledged	14

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Engagement Summary

After the initial version of the code was reviewed, this report was delivered to Primitive Finance on the 14th of June, 2022. Due to the potentially major changes in the codebase, Primitive Finance decided not to proceed with the fixes review during this engagement.

The Hyper later was renamed to Portfolio, which can be found at https://github.com/primitivefinance/Portfolio. It is important to note that the current codebase of Portfolio has been heavily changed and differs from the one that was assessed. Therefore, the content of this report is limited to the code commits explicitly mentioned in the table provided below.

2.2 Scope

The assessment was performed on the source code files inside the repositories based on the documentation files provided by Primitive Finance.

The table below indicates the code versions relevant to this report and when they were received.

Hyper

V	Date	Commit Hash	Note
1	16 May 2022	ff3c1f332a2e1ac027c905dacbd7b51084fbb023	Initial Version

rmm-examples

V	Date	Commit Hash	Note
1	16 May 2022	2dcc1190eea6f335fc6c6e9ca1ccea63ddf6e109	Initial Version

bits

V	Date	Commit Hash	Note
1	16 May 2022	b5a62af6c4d1bf5fedb35d852ba5971f588f9212	Initial Version

migrator

V	Date	Commit Hash	Note
1	16 May 2022	8334a471d6c51afc579096b41924a36f08c64057	Initial Version

For each repo from the table above, the assessment was performed on the following solidity smart contracts:

- `Hyper` repo: all contracts in the `contracts` folder, if not mentioned in Excluded from scope.
- `rmm-example` repo: all contracts in the `contracts/liquidityManager` and `contracts/liquidityWrapper` folders.
- `bits` repo: `contracts/ERC20.sol` and `contracts/ERC1155.sol` solidity smart contracts.
- `migrator` repo: `contracts/Migrator.sol` solidity smart contract.

For the `Hyper` repo solidity smart contracts, the compiler version `0.8.10` was chosen. For the `migrator` and `rmm-examples` repo solidity smart contracts, the compiler version `0.8.9` was chosen. For the `bits` repo solidity smart contracts, the compiler version `0.8.13` was chosen.

2.2.1 Excluded from scope

All files, contracts and libraries imported from external repositories are excluded from the scope, e.g. contracts from the externally imported `@primitivefi/rmm-core` and `@primitivefi/rmm-manager` repos.

In the `Hyper` repo, all smart contracts from the `contracts/test` folder were not considered as part of this assessment.

2.3 Assumptions

We assume that the reviewed contracts will interact only with external contracts that they are compatible with. In the case of ERC20 external token contracts, we assume the following:

- Token balance modifications don't happen outside of transfer functions. Particularly: no airdrops, rebasing, deflationary tokens that decrease balance, Interest bearing tokens that increase balances.
- Optional (according to EIP-20) `decimals` variable getter is implemented.
- Tokens don't have fees on transfer.

Also, the following tokens can break the protocol depending on their use:

- Tokens with blacklisting in case a Primitive contract is blacklisted
- Pausable tokens when paused
- Upgradable tokens that later introduce one of the problematic features

2.4 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

The system overview of the Previous Assessment is applicable to the current assessment as well.

Primitive's replicating market maker (RMM) is designed to replicate the payoff function of a covered call option, allowing liquidity providers to express their view on the future price of a token through different pool parameters. Each RMM pool corresponds to a token pair and a curve. The curve is characterized by strike price, maturity date, and implied volatility (sigma). Additionally, pools are subjected to a fee regime, which replicates option premium/theta decay (i.e., the decrease of an option's value over time).

RMM pools utilize arbitrage for a price alignment with a market and thus have no dependency on oracles. At expiration, assuming the pool spot price is aligned, a liquidity position will be redeemed either

1. For base tokens (if they are valued below the strike price), or
2. For an amount of quote tokens corresponding to the strike price.

Primitive Hyper is a system for executing orders optimized for optimistic rollups, like Arbitrum. On optimistic rollup chains, execution is cheap while calldata remains expensive, as it must be pushed to the L1 blockchain for data availability.

2.4.1 Primitive Hyper

Primitive Hyper RMM is a system for executing orders, optimized for optimistic rollups like Arbitrum. On optimistic rollup chains, execution is cheap while calldata remains expensive, as it must be pushed to the L1 blockchain for data availability.

The main contracts of Primitive Hyper are:

- EnigmaVirtualMachine
- HyperLiquidity
- HyperSwap
- Compiler

2.4.1.1 EnigmaVirtualMachine

The smaller calldata message size requirements in Hyper come from a novel processing machine - Enigma. Enigma is an abstract contract that defines storage fields and supported instructions. Below are the instructions supported by the Enigma VM:

- Unknown (`0x00`)
- Add Liquidity (`0x01`)
- Remove Liquidity (`0x03`)
- Swap (`0x05`)
- Create Pool (`0x0B`)
- Create Curve (`0x0C`)
- Create Pair (`0x0D`)
- Jump (`0xAA`)

Enigma can process multiple exchange operations in one transaction through the "jump" instruction. The jump instruction relies on a special encoding and pointers to separate the different operations. After the jump code byte, a single byte indicates the amount of transactions to process. The next byte is a pointer that indicates the location of next pointer. A `for` loop iterates through all the individual instructions before balances are settled in one function call at the end of the fallback function.

2.4.1.2 HyperLiquidity

HyperLiquidity is an abstract contract, which handles all pool-liquidity related interactions as well as the creation of new pools, curves and pairs.

2.4.1.3 HyperSwap

HyperSwap is an abstract contract that implements the main swap functionality that enables the arbitrage and thus alignment of the pool prices with the market.

2.4.1.4 Compiler

The Compiler smart contract is a main entry point for the Primitive Hyper system. It extends all aforementioned contracts and dispatches the functions and instructions to the internal functions. The following external state modifying functions can be called on the Compiler:

- `draw`: remove tokens from internal balance.
- `fund`: deposit tokens into internal balance.
- `updateLastTimestamp`: sync the specified pool's timestamp with the current block timestamp.

- `fallback`: dispatches the Enigma instructions and their parameters, which are sent to the Compiler as `msg.data`.

2.4.2 RMM examples

The RMM examples library is a set of smart contracts that showcase how contracts should interact with the Primitive RMM contracts. Developers that plan to integrate or build on top of the Primitive RMM system can use this library as a boilerplate or reference implementation. The two contracts that were reviewed during the assessment are: `LiquidityManager` and `LiquidityWrapper`.

2.4.2.1 LiquidityManager

`LiquidityManager` is a contract that acts as an adapter for `PrimitiveManager` of the Primitive RMM system. `LiquidityManager` can be seen as a basic version of `PrimitiveManager` that simplifies certain actions, like management of `PrimitiveManager.margin` balance. `LiquidityManager` is associated only with a unique risky-stable pair and can be used to allocate liquidity into different pools for that pair.

2.4.2.2 LiquidityWrapper

`PrimitiveManager` is an ERC1155 token that tokenises the liquidity that is allocated into the `PrimitiveEngine`. `LiquidityWrapper` contract is an ERC20, that allows wrapping defined poolId ERC1155 `PrimitiveManager` token. This allows easier interaction with external systems and removes the friction that `PrimitiveManager` might have when integrating with other protocols. The `LiquidityWrapper` ERC20 token can be unwrapped back to the `PrimitiveManager` ERC1155 tokens.

2.4.3 Bits library

Bits library is a smart-contract library inspired by the OpenZeppelin and Solmate libraries. The `ERC20` and `ERC1155` tokens defined in this library are implemented with the intention of minimising their gas usage, by performing various optimisations.

2.4.4 Migrator

The `Migrator` contract can be used to migrate liquidity from popular DEXes to Primitive RMM. In the current state, the `Migrator` has two functions: `migrateUniV2` and `migrateUniV3` allow users to migrate liquidity from Uniswap V2 and V3 pools into the Primitive RMM. With the help of these functions, users can exit a Uniswap pool and allocate liquidity into a Primitive pool in one transaction.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe our findings. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation
- Trust: Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	2

- Compiler Internal User Balances Not Accounted for in Settlement Acknowledged
- LiquidityManager Mixes the Liquidity From Different Pools Acknowledged

High-Severity Findings	7

- Compiler ABI Collisions Acknowledged
- ERC1155 Incorrect Function Selectors Acknowledged
- ERC1155 Incorrect Return Value in supportsInterface Acknowledged
- HyperLiquidity Unsafe Cast of IDs Acknowledged
- LiquidityManager Does Not Withdraw Tokens Acknowledged
- Migrator Atomic Approvals Acknowledged
- Migrator Unchecked Token Ratios When Allocating Acknowledged

Medium-Severity Findings	10

- Compiler Events Don'T Log the Msg.Sender Acknowledged
- Compiler Internal User Balances Accounting Acknowledged
- Decoder No Length Check Acknowledged
- ERC1155 Backward Batch Transfer Not Compliant With Standard Acknowledged
- ERC1155 Incorrect Origin in Mint Batch Function Acknowledged
- Function HyperSwap.getInvariant Can Fail Due to Overflow Acknowledged
- Hyper Canonical Form of Encoding Not Enforced Acknowledged
- HyperLiquidity Curve Params Do Not Define the ID Acknowledged
- LiquidityManager transferFrom Return Value Ignored Acknowledged
- Migrator Partial Migration Acknowledged

Low-Severity Findings	14

- Compiler Fallback Function Is Payable Acknowledged
- ERC1155 Transfer Events Must Be Emitted Before Calling Hook Acknowledged
- Floating Dependencies Versions Acknowledged
- Gas Optimisations Acknowledged

- HyperLiquidity Maximum Number of Pairs Acknowledged
- Incorrect Encoding Documentation Acknowledged
- Instructions Incorrect NatSpec Acknowledged
- LiquidityManager Can Run Out of Approvals Acknowledged
- LiquidityWrapper Permit Signature Replay Acknowledged
- LiquidityWrapper Unwrap Event Reentrancy Acknowledged
- Migrator Integration With Uniswap and Primitive RMM Acknowledged
- Migrator UNI V3 Fees Not Collected Acknowledged
- Multicall Payable Function Acknowledged
- RMM-examples Differing Compiler Versions Acknowledged

5.1 Compiler Internal User Balances Not Accounted for in Settlement

Security Critical Version 1 Acknowledged

In `Compiler._settleToken()`, internal token balances that are in active positions (`global`) are compared to the on-chain token balance of the contract (`actual`).

```
uint256 global = globalReserves[token];
uint256 actual = _balanceOf(token, address(this));
if (global > actual) {
    uint256 deficit = global - actual;
    _applyDebit(token, deficit);
} else {
    uint256 surplus = actual - global;
    _applyCredit(token, surplus);
}
```

If `global < actual`, the user is credited the surplus to their internal balance in `_applyCredit()`.

```
function _applyCredit(address token, uint256 amount) internal {
     balances[msg.sender][token] += amount;
     emit Credit(token, amount);
 }
```

After this, it is expected that the invariant `global == actual` holds. However, global is not updated in `_applyCredit()`. This is because internal user balances that are not in active positions are not considered part of global. This means that `global < actual` continues being true after _settleToken() completes, breaking the invariant. As a result, every time `_settleToken()` is called, the user will be credited again. This incorrectly increases their internal balance.

Example attack:

- `fund(tokenA,10)`
- Swap in a pool containing tokenA, this calls `_settleToken(tokenA)`
- `balances[msg.sender][token]` is now 20
- `draw(tokenA,20,msg.sender)`

By repeating the swap multiple times, the user can receive an arbitrarily large internal balance and drain all tokens from the contract.

Acknowledged:

See Engagement summary .

5.2 LiquidityManager Mixes the Liquidity From Different Pools

Correctness Critical Version 1 Acknowledged

LiquidityManager.allocate puts user tokens into the PrimitiveEngine via a call to PrimitiveManager.allocate. The LiquidityManager.allocate takes a user-defined parameter poolId that defines which exact curve (strike, sigma, maturity, gamma) should be used. Later, the LiquidityManager.liquidityOf[msg.sender] is increased by del liquidity. Similarly, the user can exit a pool by calling LiquidityManager.remove with the desired pool id. Nothing stops the user from entering into one poolId and exiting from the other poolId. However, the delLiquidity values for different pools can vary, due to the way PrimitiveEngine defines the liquidity unit. How much one risky/stable token liquidity unit is worth is defined during the creation of the pool in the PrimitiveEngine.create function by the riskyPerLp parameter.

Assume LiquidityManager has users who allocated liquidity in 2 pools: A and B. One with a low riskyPerLp and another with a high riskyPerLp. The attacker, by allocating liquidity into A and removing from B, can effectively "pump" the liquidity that belongs to other users from the LiquidityManager.

Acknowledged:

See Engagement summary .

5.3 Compiler ABI Collisions

Design High Version 1 Acknowledged

The interface of the Compiler contract is defined in 2 separate ways: by default Solidity ABI and by custom Enigma instructions. The execution is dispatched to the functions, if in fallback function of the Compiler, lower 4 bits of the first byte(full byte for jump instruction) are matched with some defined instruction.

```
bytes1 public constant UNKNOWN = 0x00;
bytes1 public constant ADD_LIQUIDITY = 0x01;
bytes1 public constant REMOVE_LIQUIDITY = 0x03;
bytes1 public constant SWAP = 0x05;
bytes1 public constant CREATE_POOL = 0x0B;
bytes1 public constant CREATE_PAIR = 0x0C;
bytes1 public constant CREATE_CURVE = 0x0D;
bytes1 public constant INSTRUCTION_JUMP = 0xAA;
```

However, the fallback instruction processing happens after the solidity ABI functions are matched. Hence, the instructions can accidentally be matched first with the public functions of the Compiler contracts. For example, CREATE_PAIR which uses "0x0C" byte, collides with public getter UNKNOWN() with 4 byte hash "0c78932d". In a rare case when the token base address that follows the instruction data has the needed 3 other bytes as start bytes, the creation of the pair will be not possible, because the getter will be executed instead. Similarly, CREATE_POOL "0x0B" collides with "7b1837e" `fund(address,uint256)` function.

Such collisions can be misused by malicious parties to mislead users into signing data that will lead to loss or lock of funds.

In combination with Hyper Canonical Form of Encoding not Enforced issue, this leads to other kind of collisions. For example, "e302c809" function `updateLastTimestamp(uint48)` can collide with REMOVE_LIQUIDITY. The high e 4 bytes of the first byte will be treated as a `useMax` flag by the instruction.

Acknowledged:

See Engagement summary.

5.4 ERC1155 Incorrect Function Selectors

Correctness **High** **Version 1** **Acknowledged**

In functions `safeBatchTransferFrom` and `_mintBatch`, when the `IERC1155TokenReceiver` hook is called, the return value is checked against the wrong function selector. The function selector of `onERC1155BatchReceived` should be used instead of `onERC1155Received`.

Acknowledged:

See Engagement summary .

5.5 ERC1155 Incorrect Return Value in `supportsInterface`

Correctness **High** **Version 1** **Acknowledged**

The `supportsInterface` function returns true on `0x4e2312e0`, which is incorrect as this corresponds to the `ERC1155TokenReceiver` interface. It should instead return true on `0xd9b67a26` because it implements the ERC1155 interface.

The fact that this function also returns true on `0x01ffc9a7` is correct, however, as this indicates support for the ERC165 interface. The specification states:

> Smart contracts implementing the ERC-1155 standard MUST implement the ERC-165 supportsInterface function and MUST return the constant value true if 0xd9b67a26 is passed through the interfaceID argument.

Acknowledged:

See Engagement summary .

5.6 HyperLiquidity Unsafe Cast of IDs

Security High Version 1 Acknowledged

In `_createPair()`, the `pairId` is set by casting `uint256 pairNonce` to `uint16`. This means `pairId` can silently wrap around, overwriting previously allocated pairs if `pairNonce` is larger than `2^16`.

The same issue is present in `_createCurve()`, where `uint256 curveNonce` is cast to `uint32`. Here, previously allocated curves can be overwritten if `curveNonce` is larger than `2^32`.

Acknowledged:

See Engagement summary .

5.7 LiquidityManager Does Not Withdraw Tokens

Correctness High Version 1 Acknowledged

`LiquidityManager.remove` calls `PrimitiveManager.remove` functions and then transfers `risky` and `stable` tokens to the `msg.sender`. However, by default, `PrimitiveManager.remove` will not transfer the tokens to the `LiquidityManager`. Instead, they will be added to the `PrimitiveManager.margins` balance. This margin balance needs to be explicitly claimed via the `PrimitiveManager.withdraw` function. In the current state, the transfers of tokens in the `LiquidityManager.remove` will fail.

Acknowledged:

See Engagement summary .

5.8 Migrator Atomic Approvals

Correctness High Version 1 Acknowledged

`migrateUniV3` and `migrateUniV2` both take `owner` as a parameter, which means that anyone can migrate from any LP tokens that have been approved by anyone else. This works as intended, as long as approvals and migrations happen atomically.

In `migrateUniV2`, this can be achieved as follows.

Using Multicall:

1. Call `selfPermit`
2. Call `migrateUniV2`

Note: It is important not to permit a larger amount than is being migrated.

For `migrateUniV3` this is not possible, as `Migrator` has no selfPermit function for the ERC721 tokens used in Uniswap V3, only for ERC20 tokens.

This means that any time an EOA wants to use `migrateUniV3`, they would first need to `approve` in a separate transaction. Then they could get frontrun before being able to call `migrateUniV3`, losing their funds.

Acknowledged:

See Engagement summary .

5.9 Migrator Unchecked Token Ratios When Allocating

Design High Version 1 Acknowledged

Migrator removes tokens from a Uniswap LP position, then calls `PrimitiveManager.allocate` with the full amounts received.

From Primitive's documentation: "The add liquidity function expects two amount arguments: the amount of base tokens and amount of quote tokens to provide. These amounts can mint a non-optimal amount of liquidity (which gifts the tokens to the pool) if they are not computed correctly."

As Migrator does not calculate the optimal deposit amount, it will almost always receive a non-optimal amount liquidity tokens, gifting tokens to the pool. This issue is present in `migrateUniV3` and `migrateUniV2`.

In case `migrateUniV3` is used to migrate a Uniswap V3 LP positions that is out of the current price range, only tokens of one type will be withdrawn. This will make `PrimitiveManager.allocate` revert, as one input amount will be zero.

Acknowledged:

See Engagement summary .

5.10 Compiler Events Don'T Log the Msg.Sender

Design Medium Version 1 Acknowledged

In following functions of the `Compiler` contract, the `msg.sender` is not logged.

1. `_applyCredit` function
2. `_applyDebit` function
3. `_swap` function

This complicates the integration with offchain systems and makes it difficult to monitor the state of the contract. In addition, functions `draw` and `fund` that rely on such functions don't log `msg.sender` as well.

Acknowledged:

See Engagement summary .

5.11 Compiler Internal User Balances Accounting

Design Medium Version 1 Acknowledged

First: In `Compiler`, the `_applyDebit` call triggered by `_settleTokens` tries to get `global-actual` tokens from the user. If `balances[msg.sender][token]` is less then this value, all tokens will be drawn from the user. The available `balances[msg.sender][token]` value won't be used. This increases the required amount of tokens the user must hold and the required approval the user must give to the `Compiler` contract. For example,

1. User holds 100 in `balances[msg.sender][token]`
2. User performs some operations that result in `_applyDebit` of 101 tokens.
3. All 101 tokens will be transferred from the user again. His `balances[msg.sender][token]` will remain 100.

So instead of providing only 1 token extra, the user had to provide 101. Such behavior forces users to execute more transactions to achieve the same effect.

Second: The `useMax` flag for the swap and `_addLiquidity` operations use only the full `token.balanceOf` value. All the `balances[msg.sender][token]` values are ignored. While this causes no direct problems, this adds extra transaction that a user must execute for certain scenarios.

Acknowledged:

See Engagement summary .

5.12 Decoder No Length Check

Design Medium Version 1 Acknowledged

The functions in `Decoder` take bytes arrays of arbitrary length as inputs. They do not check that the length does not exceed the expected length.

In `toBytes32`, if the length of the array is larger than 32 bytes, `shift` will overflow, leading to an incorrect shift amount. The return value will likely be zero, as the data is shifted by a very large value.

In `toBytes16`, if the length of the array is larger than 16 bytes, `shift` will overflow, leading to an incorrect shift amount. The return value will likely be zero, as the data is shifted by a very large value.

In `toAmount`, the `toBytes16` function is called with an array that can be longer than 16 bytes.

Acknowledged:

See Engagement summary .

5.13 ERC1155 Backward Batch Transfer Not Compliant With Standard

Correctness Medium Version 1 Acknowledged

In `safeBatchTransferFrom`, for gas efficiency reasons, transfers are performed backward (i.e., from the last token id to the first token id).

This is not compliant with the EIP-1155 standard, which specifies that "the balance changes and events MUST occur in the array order they were submitted."

Acknowledged:

See Engagement summary .

5.14 ERC1155 Incorrect Origin in Mint Batch Function

Correctness Medium Version 1 Acknowledged

In the `_mintBatch` function, the second argument given to `onERC1155BatchReceived` is `_to`, which is incorrect. Instead, for mint operations, this argument should be the zero address because it specifies the "address which previously owned the token".

Acknowledged:

See Engagement summary .

5.15 Function `HyperSwap.getInvariant` Can Fail Due to Overflow

Design Medium Version 1 Acknowledged

In all the functions of the `HyperSwap` contract, the `pool._updateLastTimestamp` does not increase with `blockTimestamp` after the curve maturity. However in `HyperLiquidity` contract functions `_increaseLiquidity` and `_removeLiquidity` perform the update without taking the maturity into account. Thus, by adding/removing the liquidity to the pool after the maturity, the `pool.blockTimestamp` can become greater than `curve.maturity`.

Thus, the following computation performed in the `HyperSwap.getInvariant` might overflow and revert, preventing swaps after maturity:

```
uint32 tau = curve.maturity - uint32(pools[poolId].blockTimestamp);
```

Acknowledged:

See Engagement summary .

5.16 Hyper Canonical Form of Encoding Not Enforced

Design Medium Version 1 Acknowledged

The Enigma decoder permits multiple ways of expressing the same operation. For example, any 4 bit input for `useMax` that is not exactly 1 will be treated the same way as if `useMax == 0`. In addition, the decoding of instructions does not enforce a length constraint on the input bytes. For example, decoding operations such as `Decoder.toAmount(data[7:])` can have garbage bytes or missing bytes, that are equivalent to some canonical representation of the same data, but without those bytes. Similarly, in `deltaLiquidity = uint128(bytes16(data[23:]))` padding will be ignored.

Acknowledged:

See Engagement summary .

5.17 HyperLiquidity Curve Params Do Not Define the ID

Design Medium Version 1 Acknowledged

`bytes32 rawCurveId` consists of `(uint24 sigma, uint32 maturity, uint16 fee, uint128 strike)`. However this is only 26 bytes of the data. Curves with same set of parameters will be treated as different, if the remaining 6 bytes is them are different. Thus, curve id does not uniquely define the params of the curve.

Acknowledged:

See Engagement summary .

5.18 LiquidityManager transferFrom Return Value Ignored

Correctness Medium Version 1 Acknowledged

First, `allocate` uses `IERC20.transferFrom` without checking the return value. It is possible that an ERC20 token returns false on failure to transfer, without reverting. According to ERC20 spec : "Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned!". In this specific case this is not a breaking issue, as `LiquidityManager` is not expected to hold any tokens from previous transactions. If the contract holds no tokens `IPrimitiveManager.allocate` will revert, reverting the entire transaction.

Second, some tokens do not return a bool value at all. For example, USDT, BNB, OMG. For such tokens the transfer will result in revert of entire transaction.

The `PrimitiveManager` and `PrimitiveEngine` are compitable with both kinds of tokens, thus `LiquidityManager` can operate only with a subset of tokens and pairs that original Prifimive Finance system supports.

Other functions in `LiquidityManager` also call `IERC20.transfer` and `IERC20.approve` without checking their return values, which creates similar problems.

If a `transfer` fails with a return value of false in `remove`, tokens could get stuck in the `LiquidityManager`.

Acknowledged:

See Engagement summary .

5.19 Migrator Partial Migration

Design Medium Version 1 Acknowledged

`migrateUniV3` takes a liquidity parameter. If `liquidity` is not the full liquidity of the Uniswap V3 LP position NFT, only part of the liquidity will be migrated, but the NFT associated with the `tokenId` will stay in the possession of the `Migrator` contract at the end, not returned to the user. There still can remain tokens that will not be migrated from the UniV3 pool.

Acknowledged:

See Engagement summary .

5.20 Compiler Fallback Function Is Payable

Design | Low | Version 1 | Acknowledged

Payable fallback functions allows contract to receive native Ether. However, contract has no logic to handle it. Any `msg.value` sent to the contract will be locked. Since users can perform this accidentally, having a payable fallback functions is not safe.

Acknowledged:

See Engagement summary .

5.21 ERC1155 Transfer Events Must Be Emitted Before Calling Hook

Correctness | Low | Version 1 | Acknowledged

In `safeTransferFrom` and `safeBatchTransferFrom`, an event is emitted at the end of the function.

The EIP-1155 standard specifies that "the transfer event MUST have been emitted to reflect the balance changes before the ERC1155TokenReceiver hook is called on the recipient contract."

A re-entrancy into one of these transfer functions by the recipient could lead to a wrong ordering of events.

Acknowledged:

See Engagement summary .

5.22 Floating Dependencies Versions

Design | Low | Version 1 | Acknowledged

In reviewed smart contract project repos, the versions of the contract libraries in `package.json` are not fixed. Please consider the following examples:

```
"@primitivefi/rmm-core": "^1.0.0",
"@uniswap/v3-periphery": "^1.4.0",
"@rari-capital/solmate": "^6.3.0",
```

Caret ^version will accept all future minor and patch versions, while fixing the major version. With new versions being pushed to the dependency registry, the result of compilation can change. This may lead to incompatibilities with older compiled smart contracts. For example, if imported parent contract changes the storage slot order or changes function parameter order, the children contracts might have different storage slots or different interfaces, if compared to older compiled contracts. In addition, this can lead to issues when trying to recreate the exact bytecode of a deployed contract.

Acknowledged:

See Engagement summary .

5.23 Gas Optimisations

Design Low Version 1 Acknowledged

The following optimisations can be performed to improve the gas consumption:

1. `Newton.compute` function does not need to compute the h before the loop, since it will be recomputed upon entering the loop.
2. `Compiler._applyDebit` can decrease the `balances` via unchecked block.
3. `HyperSwap._swap` writes the `pool.blockTimestamp` twice. First in the `_updateLastTimestamp` call, then in the body of the `_swap`.
4. `HyperSwap._swap` performs repetetive reads of the `pool` storage fields. E.g. `internalLiquidity`, `internalBase`, `internalQuote`.
5. `HyperSwap._swap` reads `pool.blockTimestamp` instead of using `lastTimestamp`.
6. `HyperSwap._updateLastTimestamp` will perform a write to the `pool.blockTimestamp`, even if the maturity was reached already.
7. `EnigmaVirtualMachine`, the mappings storing less than 1 storage word will have redundant bit masking operations. Converting types to full word datatypes might improve the gas consumption.
8. `LiquidityManager` fields `risky` and `stable` can be immutable.
9. `LiquidityManager.allocate` after the allocation handles the stable and risky dust tokens. However, in normal conditions, `delRisky` and `delStable` params passed to the `allocate` function will be fully used by `PrimitiveManager` and `PrimitiveEngine`, leaving no dust. The dust sweeping on each `allocate` is a gas costly operation.
10. `LiquidityWrapper` fields `manager`, `poolId` and `empty` can be immutable.
11. `Migrator` can safe gas by relying on fact that Uniswap tokens in pair are always ordered `token0 < token1`.
12. `Migrator.migrateUniV2` approves the manager in every call.

    ```
    IERC20(token0).approve(params.manager, amountA);
    IERC20(token1).approve(params.manager, amountB);
    ```

    It is expected that the same manager will be approved for the same tokens many times. As Migrator has an unprotected sweep function, any tokens remaining in the contract after a call should be seen as lost. This means it is not a problem if Migrator gives approvals that are too high. The approves in every call could be replaced by a public approve function, which is only called occasionally but with higher approval amounts.
13. `ERC20` fields `name` and `symbol` can be immutable.

14. ERC20 function `_burn` can benefit from similar to `_mint` unchecked block gas safe trick, since `totalSupply >= balanceOf >= value`.

Acknowledged:

See Engagement summary .

5.24 HyperLiquidity Maximum Number of Pairs

Design | Low | Version 1 | Acknowledged

`pairId` is stored in a `uint16`, meaning there can only be `2^16 == 65536` different pairs. A malicious actor could purposefully create pairs and exhaust the available IDs. Taking into account the HyperLiquidity Unsafe Cast of IDs issue, this will create a bad state of the system.

The same issue is present to a lesser extent for curves, where `curveId` is stored in a `uint32`. Creating `2^32` curves would be more expensive, but could be possible on chains with very low fees.

Acknowledged:

See Engagement summary .

5.25 Incorrect Encoding Documentation

Correctness | Low | Version 1 | Acknowledged

In section 3. of the documentation for the Compiler, the "Enigma encoding algorithm" section is incomplete for some instructions.

1. The documentation for encoding for "Create Curve" is missing the 16 bytes for strike.
2. The documentation for encoding for "Swap" is missing the 1 byte for direction.

Deviations of documentation from code can lead to human-errors and unexpected bugs.

Acknowledged:

See Engagement summary .

5.26 Instructions Incorrect NatSpec

Correctness | Low | Version 1 | Acknowledged

The NatSpec of Instructions is incorrect or confusing in multiple places.

1. `Instructions.decodeAddLiquidity @param data` states that the maximum length for data is 40 bytes. The correct maximum length is 42 bytes. The "power" bytes were not counted.
2. `Instructions.decodeCreateCurve @dev` states that a bytes array with length 25 is expected. This could be confusing, as the bytes array data should have length 26, including the `ecode` field.
3. `Instructions.decodeRemoveLiquidity @param data` states that the maximum length for data is 23 bytes. The correct maximum length is 24 bytes. The "power" byte was not counted.

4. `Instructions.decodeSwap` `@param data` states that the maximum length for data is 24 bytes. The correct maximum length is 42 bytes. The "power" and "amountOut" bytes were not counted. The correct format is: | 0x | 1 byte packed flag-enigma code | 6 byte poolId | 1 byte power of amountIn | up to 16 byte amountIn | 1 byte power of amountOut | up to 16 byte amountOut | 1 byte direction |

Acknowledged:

See Engagement summary .

5.27 LiquidityManager Can Run Out of Approvals

Design | Low | Version 1 | Acknowledged

The `LiquidityManager` gives (during the creation of the contract) approvals to `PrimitiveManager` to spend `uint256.max` stable and risky tokens. All allocations via this contact will use these approvals for the transfers of tokens. Over time, the initial amount can be fully used. Malicious parties can intentionally drain the approvals. Since there is no way to renew the approvals after deployment, the drained contract will not be able to allocate tokens into the pool.

Acknowledged:

See Engagement summary .

5.28 LiquidityWrapper Permit Signature Replay

Design | Low | Version 1 | Acknowledged

The `Multicall` contract and `selfPermit` functions allow `LiquidityWrapper` to approve and wrap tokens in a single transaction. However, the data passed to the `selfPermit` can be used by 3rd parties. Malicious users can frontrun the bundle and submit the `manager`, `v`, `r` and `s` values to the function directly, thus potentially causing the initial multicall bundle to revert. In comparison, the `Migrator.selfPermit` only allows the approval of `msg.sender` and not a user-defined `manager`.

Acknowledged:

See Engagement summary .

5.29 LiquidityWrapper Unwrap Event Reentrancy

Correctness | Low | Version 1 | Acknowledged

The `LiquidityWrapper.unwrap` calls `IERC1155.safeTransferFrom`, which triggers the `onERC1155Received` hook, which effectively gives control to external contract. External contracts can perform actions on other contracts, including `LiquidityWrapper`. After the `safeTransferFrom` call, the `Unwrap` event is emitted. All the events logged during the external call appear before the `Unwrap` event. While having no effects on transaction execution, such order complicates the monitoring and reconstruction of contract state based on the events info.

Acknowledged:

See Engagement summary .

5.30 Migrator Integration With Uniswap and Primitive RMM

Trust | Low | Version 1 | Acknowledged

The `Migrator` contract gets a lot of data from the user as input arguments. In some cases this can be misused by the user, by providing contracts and addresses that do not belong to Uniswap. For example, `Migrator.migrateUniV3` trusts the user to provide correct addresses for `nonfungiblePositionManager` and `manager`. `Migrator.migrateUniV2` trusts the user to provide correct addresses for `pair`, `manager` and `router`. Both for UniV2 and UniV3 the addresses are known and can be either fixed or computed using periphery libraries, using the addresses of the tokens and factory. Similary, the `PrimitiveManager` address is known.

Due to the stateless nature of the `Migrator`, the effect of passing wrong data is limited, but can be used my malicious actors as a potential attack vector on external systems.

5.31 Migrator UNI V3 Fees Not Collected

Design | Low | Version 1 | Acknowledged

In `migrateUniV3`, first `INonfungiblePositionManager.decreaseLiquidity` is called to remove the LP position and increase the internal balance. Then, `INonfungiblePositionManager.collect` is called to withdraw the internal balance as ERC20 tokens.

For the `amount0Max` and `amount1Max` parameters in `collect`, the return values of `decreaseLiquidity` are used. These include only the amounts that the liquidity position was reduced by, but excludes any fees collected. This limits the maximum amounts to be withdrawn. If the LP position has accrued any fees, the fees will remain in the internal balance and will not be withdrawn.

As a consequence, accrued fees will not be migrated. Together with Migrator Partial Migration issue, this will leave locked liquidity NFT with LP tokens not fully withdrawn.

Acknowledged:

See Engagement summary .

5.32 Multicall Payable Function

Design | Low | Version 1 | Acknowledged

`Migrator` and `LiquidityWrapper` inherit the payable `multicall` function from the `Multicall` contract. Almost any use of this function with non-zero msg.value is potentially dangerous and wrong. The `msg.value` cannot be used for any method callable from multicall, without potential security implications. Moreover, both `Migrator` and `LiquidityWrapper` don't handle any ETH and thus any ETH sent to them will not be accepted in the first place.

Acknowledged:

See Engagement summary .

5.33 RMM-examples Differing Compiler Versions

Design Low Version 1 Acknowledged

The RMM-examples repo contains contracts set to different compiler versions (0.8.6 and and 0.8.9). It is recommended to use the same version throughout a project.

Acknowledged:

See Engagement summary .

6 Open Questions

Here, we list open questions that came up during the assessment and that we would like to clarify to ensure that no important information is missing.

6.1 ERC20 Approval Events on `transferFrom`

Open Question **Version 1**

The `ERC20.transferFrom` function does not emit any event regarding the approval change. Thus, it is not possible to recover state based on Approval+Transfer events. While this is compliant with ERC20 specification, some libraries like OpenZeppelin, emit explicit `Approval` event during the `transferFrom`. On the other hand, `DAI` token does not emit such event. Is the lack of `Approval` event intended?

6.2 ERC20 Not Abstract

Open Question **Version 1**

In the ERC20 implementation of `bits_code`, as the supply mechanism must be added in a derived contract using the `_mint` function, it might be preferable to mark the `ERC20` contract as abstract.

Acknowledged:

See Engagement summary .

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Incomplete NatSpec

Note Version 1

1. `IPrimitiveManager`, which is used by `Migrator` is missing the `@param minLiquidityOut` on the `allocate` function.
2. `IMigrator` is missing the `@param liquidity` on the `MigrateUniV3Params` struct.
3. `IERC20` is missing the `@param _owner` on the `balanceOf` function.
4. `ILiquidityManager` is missing the `@param address` on the `liquidityOf` function.
5. `ERC1155` uses `@inheritdoc IERC165` on the `supportsInterface` function but `ERC1155` does not inherit from the `IERC165` interface. Moreover, `IERC165` does not have any NatSpec.

While all tags are optional, adding them will increase the documentation quality that will minimize the human error risk due to misunderstanding.

Acknowledged:

See Engagement summary .