Security Review Report
NM-0609 DiversiFi

NETHERMIND
SECURITY

(August 20, 2025)

Contents

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for DiversiFi's `ReserveManager` and `IndexToken` contracts. DiversiFi is a protocol that allows multiple assets to be deposited into a `ReserveManager` in return for an `IndexToken`, which represents the value of the assets held in custody by the smart contract. Each reserve has an "allocation" that defines the composition of its individual assets, with varying percentages, enabling different risk exposures across assets. The current implementation of DiversiFi is written under the assumption that each asset has equal value, making it particularly suitable for stablecoin-based use cases.

The audit comprises 754 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools, and (c) creation of test cases. All fixes were provided in `PR#4`, from the `nethermind-audit-fixes` branch.

Along this document, we report 11 points of attention, where one is classified as `Critical`, two are classified as `High`, one is classified as `Low`, and seven are classified as `Informational` or `Best Practices` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

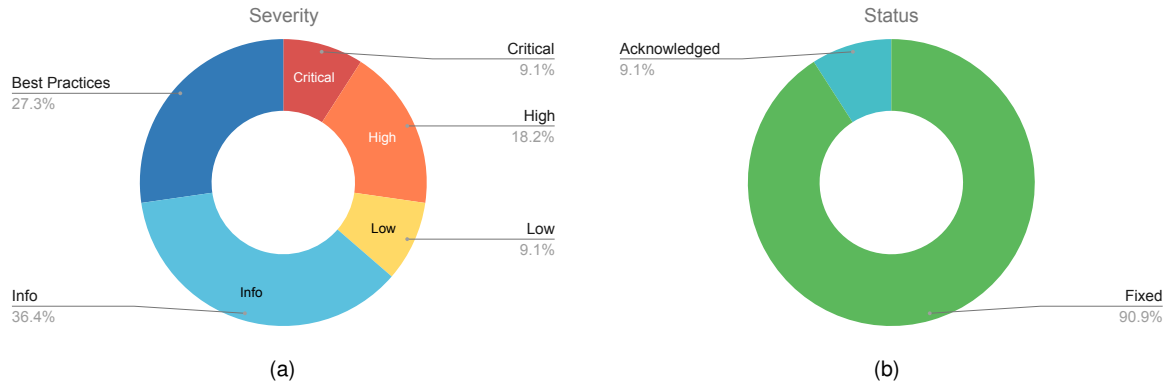


Fig. 1: Distribution of issues: Critical (1), **High** (2), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (4), **Best Practices** (3). **Distribution of status: Fixed** (10), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	August 18, 2025
Final Report	August 20, 2025
Initial Commit Hash	`29bb391bacee96efed688bcde2f414807026f473`
Final Commit Hash	`703969de56b022d2e75fdc2a7a6f313d1e71feed`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/ReserveManagerV1.sol	492	107	21.7%	73	672
2	contracts/DataStructs.sol	10	1	10.0%	3	14
3	contracts/IndexToken.sol	169	25	14.8%	39	233
4	contracts/ReserveMath.sol	83	38	45.8%	17	138
	Total	**754**	**171**	**22.7%**	**132**	**1057**

3 Summary of Issues

	Finding	Severity	Update
1	The `equalizeToTarget(...)` function allows for repeated bounty claims	Critical	Fixed
2	Allocation changes can cause value drain via rounding in `equalizeToTarget`	High	Fixed
3	Array length mismatch in `withdrawAll` can permanently block emigration and lock funds	High	Fixed
4	Zero allocation assets may not be removed from the `currentAssetParamsList_`	Low	Fixed
5	Incorrect validation for balance divisor change	Info	Fixed
6	Missing check for duplicate assets in `setTargetAssetParams`	Info	Fixed
7	Unnecessary return array in functions `mint` and `burn`	Info	Fixed
8	`ReserveManagerV1` constructor admin argument must be caller	Info	Fixed
9	ERC20 transfers should use `SafeERC20`	Best Practices	Fixed
10	Fee-on-transfer tokens can cause reserve inaccuracies	Best Practices	Acknowledged
11	Openzeppelin library is out of date	Best Practices	Fixed

4 System Overview

The DiversiFi protocol consists of two contracts: the `ReserveManager` and the `IndexToken`. The `ReserveManager` is where the underlying assets flow in and out of the contract. Each reserve has an "allocation," which is a collection of the underlying assets with varying "target allocations" that add up to 100%. For example, USDT may have a 25% allocation, DAI 25%, and USDC 50%. The `ReserveManager` aims to ensure that the actual amount of deposited tokens matches the target allocations at all times. This is achieved by adjusting token inflows and outflows during `mints` and `burns`, withdrawing single assets in exchange for `IndexTokens`, and swapping assets with users to exactly match asset amounts with the target allocations in exchange for a bounty.

4.1 Migration Process

The protocol implements a gradual migration mechanism to transition from one reserve manager implementation to another for the purpose of upgrading or recovering from a severe underlying depeg while preserving peg parity through controlled rebasement.

4.1.1 Migration Initiation

When migration begins, the current reserve manager enters *emigration mode* and signals the index token contract to start migration mode. The token contract records the initial balance divisor and begins increasing it at a specified rate per second after a configured delay period. This gradual reduction effectively rebases all token holder balances downward, ensuring the circulating supply remains fully backed by available reserves. The rebasement operates continuously and deterministically, creating a controlled deflation of the visible token supply that directly corresponds to the rate at which reserves are expected to be transferred out of the emigrating pool.

4.1.2 Reserve Migration

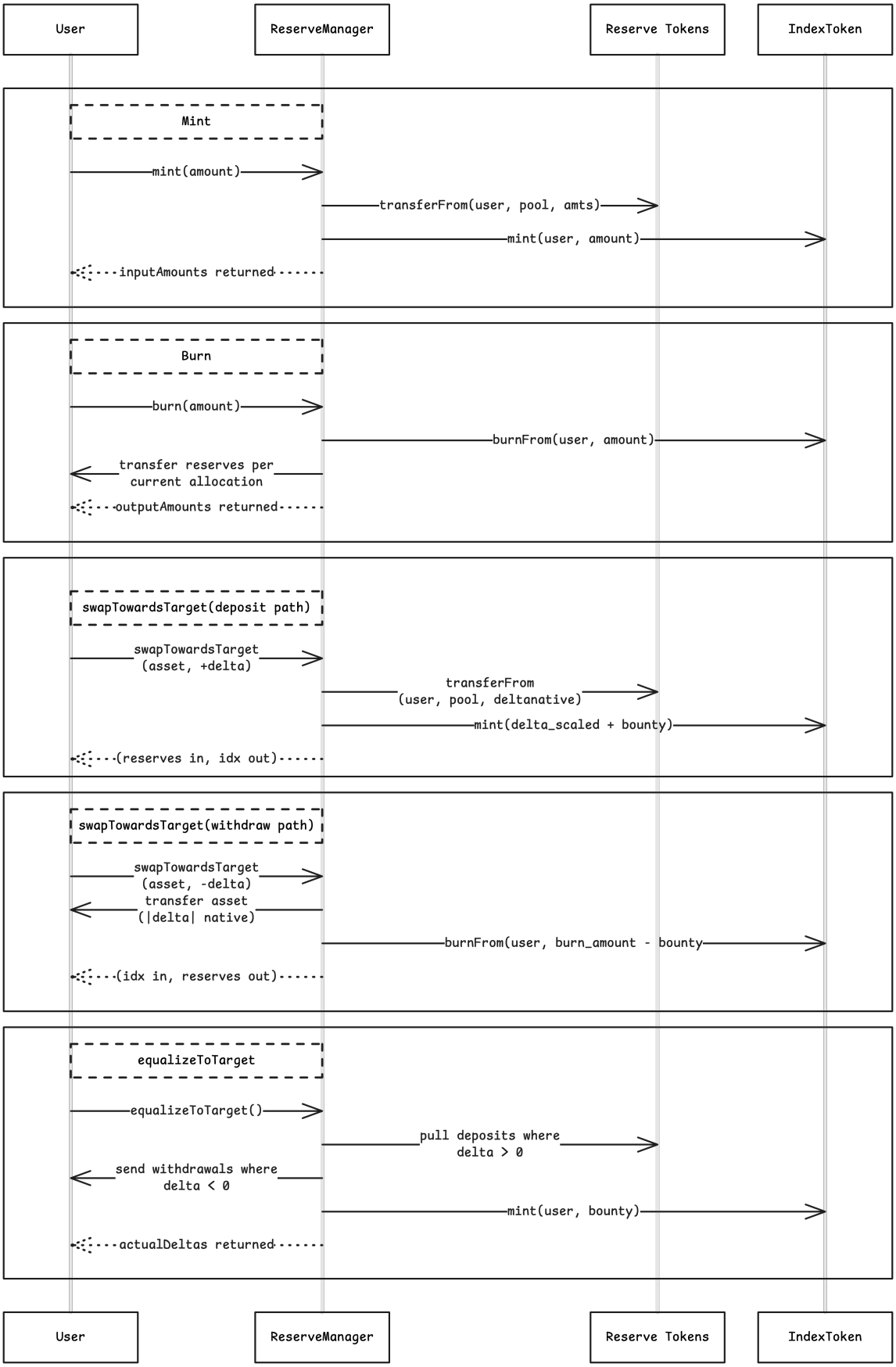
The emigrating pool auctions its reserves to the new reserve manager at a conversion rate that tracks the current balance divisor relative to its initial value. As the balance divisor increases over time, the conversion rate also increases, incentivizing faster migration and ensuring that tokens are redeemable for their appropriate share of reserves even if they are rebasing downwards. The new reserve manager operates in immigration mode, accepting reserve transfers in exchange for index tokens.

4.1.3 Migration Completion

Migration concludes when the old reserve manager has transferred all its reserves and the total reserves reach zero. The final balance divisor is computed to ensure the circulating token supply is fully collateralized by the reserves in the new reserve manager. If the new reserve manager has surplus reserves beyond what is needed for full backing, the surplus is carried over to the new pool. The index token contract then switches to point at the new liquidity pool, completing the migration.

4.2 Protocol Interaction Diagram

The following diagram visualizes the function call flows and interactions between both DiversiFi and external ERC20 contracts:



5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] The `equalizeToTarget(...)` function allows for repeated bounty claims

File(s): `contracts/ReserveManagerV1.sol`

Description: The function `equalizeToTarget` is intended to allow a user to rebalance the pool's assets to match the target allocations. As a reward for performing this action, the user receives a bounty of index tokens. The function sends the entire `equalizationBounty_` to the caller without updating the state variable `equalizationBounty_` to a new value. Anyone can call `equalizeToTarget` repeatedly and collect the same `equalizationBounty_` amount each time, effectively minting an unlimited number of index tokens, as shown in the code snippet below:

```
function equalizeToTarget() ... returns (int256[] memory) {
  // ...
  // @audit `equalizationBounty_` is not reset after being minted
  //         allowing for repeated claims.
  indexToken_.mint(msg.sender, equalizationBounty_);
  emit Equalization(deltasScaled);
  return actualDeltas;
}
```

Recommendation(s): Consider updating the `equalizationBounty_` state variable after the bounty has been paid out.

Status: Fixed

Update from the client: Fixed in commit `86ae9820`. The equalization bounty is reset to zero after the bounty is sent.

6.2 [High] Allocation changes can cause value drain via rounding in `equalizeToTarget`

File(s): `contracts/ReserveManagerv1.sol`

Description: After a pool's target allocations have been changed via `setTargetAssetParams(...)` (for example by shifting the pool weight to a low-decimal asset (e.g., 6-decimal USDC) from a high-decimal asset (e.g., 18-decimal DAI)), when `equalizeToTarget()` is called to execute this rebalance, it calculates the required deposit of the low-decimal asset. In small pools, specifically where asset reserves are less than `10` to the power of the decimal difference (e.g., `10^(18-6)`) the required deposit amount rounds down to zero during the decimal scaling process.

The function executes a `transferFrom` for zero tokens but updates its internal reserves (`specificReservesScaled_`) as if the non-zero scaled deposit was successful. The high-decimal asset is simultaneously withdrawn, real value leaves the pool without a corresponding deposit. An attacker can exploit this by repeatedly triggering such rebalances on small, mixed-decimal pools, systematically draining assets and causing the pool's internal accounting to become increasingly disconnected from its actual holdings.

```
function equalizeToTarget() ... {
  int256[] memory deltasScaled = getEqualizationVectorScaled();
  // ...
  for (uint i = 0; i < currentAssetParamsList_.length; i++) {
    AssetParams memory params = currentAssetParamsList_[i];
    if (deltasScaled[i] > 0) {
      // @audit In small, mixed-decimal pools, this can round `actualDeposit` down to 0.
      uint256 actualDeposit = PoolMath.scaleDecimals(uint256(deltasScaled[i]), DECIMAL_SCALE, params.decimals);
      // @audit The transfer of 0 tokens will succeed.
      IERC20(params.assetAddress).transferFrom(msg.sender, address(this), actualDeposit);
      // @audit `specificReservesScaled_` is updated with the non-zero `deltasScaled[i]`,
      // creating a conservation violation as no actual tokens were received.
      specificReservesScaled_[params.assetAddress] += uint256(deltasScaled[i]);
      actualDeltas[i] = int256(actualDeposit);
    } else {
      // ... (A withdrawal here can succeed, causing a net loss)
    }
  }
  // ...
}
```

Recommendation(s): Consider adding a check to prevent updates to reserves when no actual tokens are transferred or change the logic to round up in the calculations.

Status: Fixed

Update from the client: Fixed in commit `0eff82ae`. Rounding logic has been adjusted in `equalizeToTarget`.

6.3 [High] Array length mismatch in `withdrawAll` can permanently block emigration and lock funds

File(s): contracts/ReserveManagerv1.sol

Description: The `ReserveManagerv1` contract has a migration (emigration) process that allows an admin to move all assets to a new pool. This process is initiated by `startEmigration(...)` and finalized by `finishEmigration()`. A critical requirement for `finishEmigration()` is that all asset reserves in the old pool must be withdrawn, which is accomplished by calling the `withdrawAll()` function.

A severe vulnerability exists in the `withdrawAll()` function. The function initializes its `outputAmounts` return array with a size equal to `targetAssetParamsList_.length`. However, it then proceeds to loop through all assets based on the length of `currentAssetParamsList_`. If the number of assets in the current list is greater than the number of assets in the target list, the loop will attempt to write to an out-of-bounds index in the `outputAmounts` array, causing the transaction to revert.

This state can be triggered if an administrator calls `setTargetAssetParams(...)` to reduce the number of tracked assets but then calls `startEmigration(...)` *before* the `equalizeToTarget()` function is called to synchronize the `currentAssetParamsList_`.

Once emigration has started, the contract is locked in a state where `equalizeToTarget()` and `setTargetAssetParams(...)` cannot be called due to the `mustNotEmigrating` modifier. The only way to complete the migration is to call `finishEmigration()`, which requires a successful call to `withdrawAll()`. Since `withdrawAll()` is now guaranteed to revert, this creates a deadlock. The migration can never be completed, and all assets held within the `ReserveManagerv1` contract become permanently locked.

This issue is exacerbated by the iteration flaw in `equalizeToTarget()`, which can also lead to a mismatch between the current and target asset list lengths even if an admin attempts to follow the correct procedure, making this deadlock scenario more likely.

```
function withdrawAll() ... returns (AssetAmount[] memory outputAmounts) {
  indexToken_.burnFrom(msg.sender, totalReservesScaled_);
  totalReservesScaled_ = 0;
  // @audit The `outputAmounts` array is sized based on the target asset list.
  outputAmounts = new AssetAmount[](targetAssetParamsList_.length);
  uint256[] memory scaledReservesList = new uint256[](
    currentAssetParamsList_.length
  );

  // @audit The loop iterates based on the current asset list.
  for (uint i = 0; i < currentAssetParamsList_.length; i++) {
    AssetParams memory params = currentAssetParamsList_[i];
    uint256 withdrawalAmount = IERC20(params.assetAddress).balanceOf(
      address(this)
    );
    IERC20(params.assetAddress).transfer(msg.sender, withdrawalAmount);
    AssetAmount memory assetAmount;
    assetAmount.assetAddress = params.assetAddress;
    assetAmount.amount = withdrawalAmount;
    // @audit If `currentAssetParamsList_.length > targetAssetParamsList_.length`,
    // this line will cause an out-of-bounds error, reverting the transaction.
    outputAmounts[i] = assetAmount;
    specificReservesScaled_[params.assetAddress] = 0;
    scaledReservesList[i] = 0;
  }

  emit Burn(msg.sender, totalReservesScaled_, scaledReservesList, 0);
  return outputAmounts;
}
```

Recommendation(s): Consider modifying the `withdrawAll()` function to not depend on `targetAssetParamsList_`.

Status: Fixed

Update from the client: Fixed in commit `8600e13f` .The length of the `outputAmounts` array matches the iterations in the `for` loop.

6.4 [Low] Zero allocation assets may not be removed from `currentAssetParamsList_`

File(s): `contracts/ReserveManagerV1.sol`

Description: The `equalizeToTarget(...)` function is intended to rebalance the assets in the pool. It also removes assets from the `currentAssetParamsList_` if their `targetAllocation` is `0`. This is handled in a `for` loop that iterates through the list and uses `currentAssetParamsList_.pop()` to remove assets.

The issue is that the loop's iterator `i` is incremented after each iteration, but `currentAssetParamsList_.pop()` also changes the array's length and shifts the elements. If two adjacent assets have a `targetAllocation` of `0`, the first one will be popped, but the next asset will move into its place and be at the current iterator position. The `for` loop will then increment `i` and skip the newly moved asset, leaving it in the array. This can lead to a stale state where assets with zero allocation are not properly removed.

```
function equalizeToTarget() ... returns (int256[] memory) {
  int256[] memory deltasScaled = getEqualizationVectorScaled();
  int256[] memory actualDeltas = new int256[](deltasScaled.length);
  // ...
  for (uint i = 0; i < currentAssetParamsList_.length; i++) {
    AssetParams memory params = currentAssetParamsList_[i];
    if (params.targetAllocation == 0) {
      delete assetParams_[params.assetAddress];
      for (uint j = i; j < currentAssetParamsList_.length - 1; j++) {
        currentAssetParamsList_[j] = currentAssetParamsList_[j + 1];
      }
      // @audit The `i` index is not decremented to account for the pop.
      currentAssetParamsList_.pop();
    }
  }
  // ...
}
```

Recommendation(s): Consider re-implementing the logic for removing items from the array to account for the scenario described.

Status: Fixed

Update from the client: Fixed in commit `23a448d8`. A temporary array is created allowing the `currentAssetParamsList` to be deleted and refilled excluding zero target allocation assets.

6.5 [Info] Incorrect validation for balance divisor change

File(s): `contracts/IndexToken.sol`

Description: In the `IndexToken` function `startMigration` a series of validations are run on the provided arguments. The argument `balanceDivisorChangePerSecondQ96` is checked against a maximum specified value, but this comparison is done using a >= check, effectively forcing the value to be above the maximum . The relevant code is shown below:

```
function startMigration(
  ...,
  uint104 balanceDivisorChangePerSecondQ96
) external onlyReserveManager migrationCheck(false) {
  // ...
  require(
    balanceDivisorChangePerSecondQ96 >= _maxBalanceDivisorChangePerSecondQ96,
    "balance multiplier change rate too high"
  );
  // ...
}
```

It should also be noted that the argument is named as a "divisor" but the error message refers to the argument as a "multiplier", which also may need correcting.

Recommendation(s): Consider adjusting the input validation to ensure that the divisor is within the bounds of the specified maximum.

Status: Fixed

Update from the client: Fixed in commit `2ce350bc`. The check has been changed from >= to <=.

6.6 [Info] Missing check for duplicate assets in `setTargetAssetParams`

File(s): contracts/ReserveManagerv1.sol

Description: The `setTargetAssetParams(...)` function allows an admin to set the underlying assets and their target weightings for the pool. The function accepts an array of `AssetParams` but does not validate that each asset address within the array is unique.

If an admin accidentally provides an array containing duplicate asset addresses, the `targetAssetParamsList_` will store these duplicates. Core functions like `mint(...)` iterate over this list to calculate deposit amounts.

```
function setTargetAssetParams(AssetParams[] memory _params) ... {
  delete targetAssetParamsList_;
  uint88 totalTargetAllocation = 0;
  {
    // ...
    for (uint i = 0; i < _params.length; i++) {
      require(
        _params[i].assetAddress != address(indexToken_),
        "index not allowed"
      );
      // ...
      assetParams_[_params[i].assetAddress] = _params[i];
      targetAssetParamsList_.push(_params[i]);
      totalTargetAllocation += _params[i].targetAllocation;
      insertOrReplaceCurrentAssetParams(_params[i]);
    }
    // ...
  }
  // ...
}
```

Recommendation(s): Consider adding a validation step inside the `setTargetAssetParams(...)` function to ensure all asset addresses in the input array are unique. A common way to achieve this is by using a temporary mapping or a second loop to track which addresses have already been added.

Status: Fixed

Update from the client: Fixed in commit `384eff9e`. A duplicate check has been added.

6.7 [Info] Unnecessary return array in functions `mint` and `burn`

File(s): contracts/ReserveManagerV1.sol

Description: The functions `mint` and `burn` both declare a return array of type `AssetAmount` named `inputAmounts` and `outputAmounts` respectively. As each asset is iterated through a loop, and the token inflow or outflow is tracked and stored in this array. The result of this array is returned to the user, but is unlikely to provide any value as the caller of these functions are most likely to be externally owned accounts which will not be able to process this return data. A code snippet from the mint function is shown below:

```
function mint(...) ... returns (AssetAmount[] memory inputAmounts) {
  // ...
  inputAmounts = new AssetAmount[](targetAssetParamsList_.length);
  for (uint i = 0; i < targetAssetParamsList_.length; i++) {
    // ...
    AssetAmount memory assetAmount;
    assetAmount.assetAddress = params.assetAddress;
    assetAmount.amount = trueDeposit;
    inputAmounts[i] = assetAmount;
    // ...
  }
  // ...
}
```

Information about the true deposit or withdrawal amounts could be determined using the ERC20 transfer event emissions or in the case of on-chain execution where logs cannot be observed, before and after balance checks by a smart contract requiring this data could be used to determine transferred amounts.

Recommendation(s): If the return arrays `inputAmounts` and `outputAmounts` are not a technical requirement consider removing these unnecessary returns.

Status: Fixed

Update from the client: Fixed in commit `384eff9e`. The return arrays have been removed from the functions.

6.8 [Info] `ReserveManagerV1` constructor admin argument must be caller

File(s): `contracts/ReserveManagerV1.sol`

Description: The constructor for the `ReserveManagerV1` contract has the argument `_admin` which will be given the `ADMIN_ROLE` and allow access to priveleged functions. The logic flow in the constructor is to assign the role to the `_admin` address, and then call `setTargetAssetParams` to setup the assets and target allocations, as shown below:

```
constructor(address _admin, ..., AssetParams[] memory _assetParams) {
  //...
  _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
  _setupRole(ADMIN_ROLE, _admin);
  setTargetAssetParams(_assetParams);
  //...
}
```

The `setTargetAssetParams` function has the modifier `onlyRole(ADMIN_ROLE)` which requires the caller to have the `ADMIN_ROLE`. Since this role is assigned to the `_admin` argument, and the caller must have this role in order to pass the role check, the `_admin` argument can only ever be the same address as the caller.

```
function setTargetAssetParams(...) public onlyRole(ADMIN_ROLE) ... {...}
```

Recommendation(s): If the intended behavior is to have the caller as the admin, consider removing the `admin_` argument from the constructor and using `msg.sender` to determine the admin address. Alternatively if the intended behavior is to allow the admin address to be different from the deployer, consider temporarily assigning the admin role to `msg.sender` and then revoking the role, and assigning it to the `admin_` argument after `setTargetAssetParams` has been called.

Status: Fixed

Update from the client: Fixed in commit `5d226d86`. The deployer address can now be different from `_admin` arg.

6.9 [Best Practices] ERC20 transfers should use `SafeERC20`

File(s): `contracts/ReserveManagerV1.sol`

Description: The ERC20 specification states that the return value from a `transfer` or `transferFrom` should contain a boolean indicating the success or failure of a transfer. Some tokens do not adhere to this standard, such as the USDT token, which is relevant to a stablecoin focused protocol. While the `ReserveManagerV1` contract does not check return values from transfers so no real risk is present, as a best practice it is still recommended to use the `SafeERC20` library when handling token transfers.

This will also protect against some (uncommon) tokens which don't revert on failure, but return false instead. A potentially relevant stablecoin token in this case would be the Stasis EURS Token which would cause internal accounting issues in the `ReserveManagerV1` when the logic behaves as if the transfer executed successfully even though it would have failed.

Recommendation(s): Consider implementing the `SafeERC20` library on token transfers.

Status: Fixed

Update from the client: Fixed in commit `07d84ddd`. All ERC20 transfers use the `SafeERC20` library.

6.10 [Best Practices] Fee-on-transfer tokens can cause reserve inaccuracies

File(s): contracts/ReserveManagerv1.sol

Description: The contract updates its internal reserve balances based on the expected amount in token transfers, rather than the actual amount received. This is problematic for fee-on-transfer ERC20 tokens, which deduct a fee from the transfer amount.

Consequently, the contract's tracked reserves (`specificReservesScaled_`) will not match its true token balances when interacting with such tokens.

```
function mint(...) ... {
  // ...
  for (uint i = 0; i < targetAssetParamsList_.length; i++) {
    // ...
    IERC20(params.assetAddress).transferFrom(msg.sender, address(this), trueDeposit);
    // ...
    // @audit Reserves are updated based on the expected amount, not the actual
    // amount received, which will be less for fee-on-transfer tokens.
    specificReservesScaled_[
        targetAssetParamsList_[i].assetAddress
    ] += trueDepositScaled;
  }
  // ...
}
```

Recommendation(s): Assess whether new tokens to be added using `setTargetAssetParams` do not have fee-on-transfer features. Alternatively, consider calculating the actual amount transferred by checking the contract's token balance before and after every transfer.

Status: Acknowledged

Update from the client: We have decided not to fix and instead will ensure that fee-on-transfer tokens are not added as target assets.

6.11 [Best Practices] Openzeppelin library is out of date

File(s): contracts/*

Description: The `openzeppelin-contracts` library being used in the project is out of date, and contains some draft implementations which are used by the code in the DiversiFi project. An example of this is the `IndexToken` which implements the `draft-ERC20Permit.sol` file, which has since been updated to a full release. As a best practice it is recommended to use a more recent stable release of the `openzeppelin-contracts` library to avoid using draft implementations which will be inherited into the codebase.

Recommendation(s): Consider using a more recent `openzeppelin-contracts` package.

Status: Fixed

Update from the client: Fixed in commit `4b1e3320`. The Openzeppelin library has been updated.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Sygnum's documentation

`DiversiFi's` team provided an overview of the core system components during the kick-off call, presenting a clear explanation of the intended functionalities and sharing a written specification covering the different contracts and roles. In addition, the team addressed all questions raised by the Nethermind Security team, offering valuable insights and ensuring a comprehensive understanding of the project's technical foundations.

As supporting resources, the whitepaper outlined the protocol's overall design, economic rationale, and security considerations, providing the theoretical framework behind DiversiFi's mechanisms. A token migration state diagram was also provided which detailed the practical migration process for liquidity pools and governance, describing the system's state transitions across initialization, migration grace period, active migration, and finalization, including how minting, burning, and balance multipliers behave at each stage.

8 Test Suite Evaluation

8.1 Compilation Output

```
user@machine v1-contracts % npx hardhat compile
Compiled 36 Solidity files successfully (evm target: paris).

   Solc version: 0.8.27   ·  Optimizer enabled: true       ·  Runs: 100000
·····························|·····························|·····························
   Contract Name           ·  Deployed size (KiB) (change)  ·  Initcode size (KiB) (change)
   ReserveMath             ·                 0.084 ()       ·                 0.138 ()
   Address                 ·                 0.084 ()       ·                 0.138 ()
   Counters                ·                 0.084 ()       ·                 0.138 ()
   Strings                 ·                 0.084 ()       ·                 0.138 ()
   ECDSA                   ·                 0.084 ()       ·                 0.138 ()
   SignedMath              ·                 0.084 ()       ·                 0.138 ()
   Faucet                  ·                 0.429 ()       ·                 0.477 ()
   MultiMinter             ·                 1.197 ()       ·                 1.636 ()
   ReserveMathWrapper      ·                 1.935 ()       ·                 1.965 ()
   ERC20                   ·                 2.844 ()       ·                 3.539 ()
   MintableERC20           ·                 3.203 ()       ·                 3.946 ()
   ReserveManagerHelpers   ·                 6.646 ()       ·                 6.912 ()
   IndexToken              ·                 8.990 ()       ·                10.128 ()
   TimelockController      ·                 9.490 ()       ·                10.678 ()
   DFITimelockController   ·                11.843 ()       ·                13.297 ()
   ReserveManagerV1        ·                23.854 ()       ·                28.783 ()
```

8.2 Tests Output

```
IndexToken
  Special Functionality
    isMigrating
       returns false if not migrating
       returns true if migrating
    getNextReserveManager
       returns 0 address if not migrating
       returns next address if migrating
    getlastBalanceDivisor
       returns last balance multiplier
    getMigrationStartTimestamp
       returns the start timestamp that the migration started
    getBalanceDivisorChangeDelay
       should return the change delay
    getBalanceDivisorChangePerSecondQ96
       returns the balance multiplier change per second set when the migration started
    getReserveManager
       returns the current liquidity pool
    startMigration
       should not be callable by non-admin address
       should not be callable if a migration is already happening
       should fail if the change delay is too below the minimum
       should fail if the balance multiplier change per second is greater than the max rate
       should set the relevant variables
    finishMigration
       should not be callable by a non-admin
       should not be callable if not migrating
       should set the relevant variables
    balanceDivisor
       balance multiplier should be static during non-migration
       balance multiplier should tick down at the correct rate during migration
       balance multiplier should not tick down during grace period
```

```
  ERC20 Functionality
    name
       should return the name
    symbol
       should return the symbol
    increaseAllowance
       should increase the allowance
    decreaseAllowance
       should decrease the allowance
       should revert if trying to decrease allowance below zero
    totalSupply
       should return the correct value
       should change when the balance multiplier changes
       should increase when minting
       should decrease when burning
    balanceOf
       should return the correct value
       should change when the balance multiplier ticks down
    transfer
       should have no transfer inconsistencies due to rounding errors
       should not affect total supply
       should not affect allowance
    allowance
       should return the allowance in visible units
       allowance should not tick down with the balance multiplier
    approve
       should set the allowance with the correct values
    transferFrom
       should transfer the correct amount
       should deduct the allowance by the correct amount
    decimals
       should return the correct decimals
    mint
       should mint the correct amount and adjust total supply appropriately
       should still work after a balance multiplier change
    burnFrom
       should burn the correct amount
       should still work after a balance multiplier change

ReserveManager - Admin Functions
  setMintFeeQ96
     sets new mint fee and emits event when called by admin
     reverts when called by non-admin
     reverts when called during migration
  setBurnFeeQ96
     sets new burn fee and emits event when called by admin
     reverts when called by non-admin
     reverts when called during migration
  setMaxReservesIncreaseCooldown
     sets new cooldown and emits event when called by maintainer
     reverts when called by non-maintainer
     reverts when called during migration
  setMaxReservesIncreaseRateQ96
     sets new rate and emits event when called by maintainer
     reverts when called by non-maintainer
     reverts when called during migration
  setMaxReserves
     sets new max reserves, emits event, and updates timestamp when called by maintainer
     reverts when called by non-maintainer
     reverts when called during migration
  setTargetAssetParams
     sets new asset params and emits event(s) when called by admin
     reverts if an underling asset is the index token
     reverts if an underlying asset specifies incorrect decimals
     reverts if total target allocation is above 1
     reverts if total allocation is below 1
     reverts when called by non-admin
     reverts when called during migration
  withdrawFees
    - withdraws fees and emits event when called by admin
    - reverts when called by non-admin
    - reverts when called during migration
```

```
  setIsMintEnabled
     sets mint enabled and emits event when called by maintainer
     reverts when called by non-admin
     reverts when called during migration
  increaseEqualizationBounty
     should fail if the pool doesn't have enough fees collected
     should fail in the case that the balance is greater than the bounty increase, but fees collected are not
     should add the equalization bounty to the previous equalization bounty
     reverts when called during migration
  startEmigration
     should fail if the pool is already migrating
     should set all of the relevant variables
     admin functions should be disallowed while emigrating
  finishEmigration
     should fail if the pool is not currently migrating
     should fail if there are still reserves in the pool
     should set all of the relevant variables

ReserveManager - Getters
   Deployments
   getMaxReserves
   getMaxReservesIncreaseRateQ96
   getMintFeeQ96
   getIsMintEnabled
   getIndexToken
   getAllAssets
   getCurrentAssetParams
   getTargetAssetParams
   getAssetParams
   getSpecificReservesScaled
   getTotalReservesScaled
   getSpecificReserves
   getMaxReserves
   getMaxReservesIncreaseCooldown
   getLastMaxReservesChangeTimestamp
   getEqualizationVectorScaled
   getTotalReservesDiscrepancyScaled
   getIsEqualized
   getEqualizationBounty
  getBurnFeeQ96
     should return the burn fee
     should return zero if the pool is migrating
  getSurplus
     should return the token balance as fees
     should deduct the equalization bounty from the fees collected
  getMigrationBurnConversionRateQ96
     should return 1 if there is no migration
     should return an increasing number if migrating
  isEmigrating
     should return false if not emigrating
     should return true if emigrating

ReserveManager - Mint/Burn Functions
  mint
     mints liquidity tokens and updates reserves as expected
     reverts if minting is disabled
     succeeds when minting exactly up to the maxReserves limit (cooldown active)
     reverts when minting above the maxReserves limit (cooldown active)
     succeeds when minting exactly up to the NEXT maxReserves limit (cooldown inactive)
     reverts when minting above the NEXT maxReserves limit (cooldown inactive)
     reverts if reserve manager is emigrating
     succeeds if pool is being immigrated into
  burn
     burns liquidity tokens and returns assets as expected
     should give a discount if migrating
     reverts if user tries to burn more than their balance
     succeeds if pool is emigrating
     fails if pool is being immigrated into
```

```
  swapTowardsTarget
    swap token equal to standard decimal scale
      Withdraw
         should not allow swapping if the pool is equalized
         should not allow swapping that increases discrepency
         should not allow swapping that passes the target allocation
         should swap exactly to the target
         should apply an equalization bounty if one is set
         should set the bounty exactly to the burn amount if it is greater than the burn amount
      Deposit
         should not allow swapping that increases discrepency
         should not allow swapping that passes the target allocation
         should not allow a deposit that exceeds the max reserves limit
         should swap exactly to the target
         should apply an equalization bounty if one is set
    swap token below standard decimal scale
      Withdraw
         should not allow swapping if the pool is equalized
         should not allow swapping that increases discrepency
         should not allow swapping that passes the target allocation
         should swap exactly to the target
         should apply an equalization bounty if one is set
         should set the bounty exactly to the burn amount if it is greater than the burn amount
      Deposit
         should not allow swapping that increases discrepency
         should not allow swapping that passes the target allocation
         should not allow a deposit that exceeds the max reserves limit
         should swap exactly to the target
         should apply an equalization bounty if one is set
    swap token above standard decimal scale
      Withdraw
         should not allow swapping if the pool is equalized
         should not allow swapping that increases discrepency
         should not allow swapping that passes the target allocation
         should swap exactly to the target
         should apply an equalization bounty if one is set
         should set the bounty exactly to the burn amount if it is greater than the burn amount
      Deposit
         should not allow swapping that increases discrepency
         should not allow swapping that passes the target allocation
         should not allow a deposit that exceeds the max reserves limit
         should swap exactly to the target
         should apply an equalization bounty if one is set
  equalizeToTarget
     equalizes the pool
     removes zero allocation assets from assetParams_ map and currentAssetParams_ list
     applies reserves tracking and token transfers correctly
     distributes the entire equalization bounty to the caller
  withdrawAll
     should be able to withdraw all reserves
     should not be callable if the reserve manager is not emigrating

PoolMath
  allocationToFixed
     should produce the expected number
     math check
  fixedToAllocation
     should convert back to original allocation
  toFixed
     should convert to the expected number
  fromFixed
     should convert to and from fixed
  calcCompoundingFeeRate
     should calculate the correct compounding fee rate
  scaleDecimals
     same amount of decimals
     increase decimals
     decrease decimals
```

```
  calcMaxIndividualDelta
     should calculate the correct delta for a withdrawal from 0.5 to 0.25
     should calculate the correct delta for a withdrawal from 0.5 to 0
     should calculate the correct delta for a deposit from 0.25 to 0.5
     should calculate the correct delta for a deposit from 0 to 0.5
  calcEqualizationBounty
     should return 0 if the bounty is 0
     should error if the discrepency is increasing
     should return half the bounty for resolving half the discrepency
     should return the entire bounty for resolving the entire discrepency

allocationChange - complete lifecycle
   add an asset
   remove an asset
   add an asset during normal allocation change
   remove an asset during normal allocation change
   normal allocation change while adding an asset
   normal allocation change while removing an asset
   add an asset while adding an asset
   add an asset while removing an asset
   remove an asset while adding an asset
   remove an asset while removing an asset

migration - complete lifecycle
   normal migration - test that everything is as expected throughout the migration lifecycle
   multiple migrations
  migration during allocation change
     add an asset
     remove an asset
     add an asset during normal allocation change
     remove an asset during normal allocation change
     normal allocation change while adding an asset
     normal allocation change while removing an asset
     add an asset while adding an asset
     add an asset while removing an asset
     remove an asset while adding an asset
     remove an asset while removing an asset
  allocation change during migration
     add an asset
     remove an asset
     add an asset during normal allocation change
     remove an asset during normal allocation change
     normal allocation change while adding an asset
     normal allocation change while removing an asset
     add an asset while adding an asset
     add an asset while removing an asset
     remove an asset while adding an asset
     remove an asset while removing an asset

normal minting and burning activity
   minting and burning equal amounts should never result in less totalReserves than totalSupply

209 passing (1s)
3 pending
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our cryptography Research team conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.