Security Review Report
NM-0257 Chi Protocol

NETHERMIND
SECURITY

(Jul 22, 2024)

Contents

1 Executive Summary

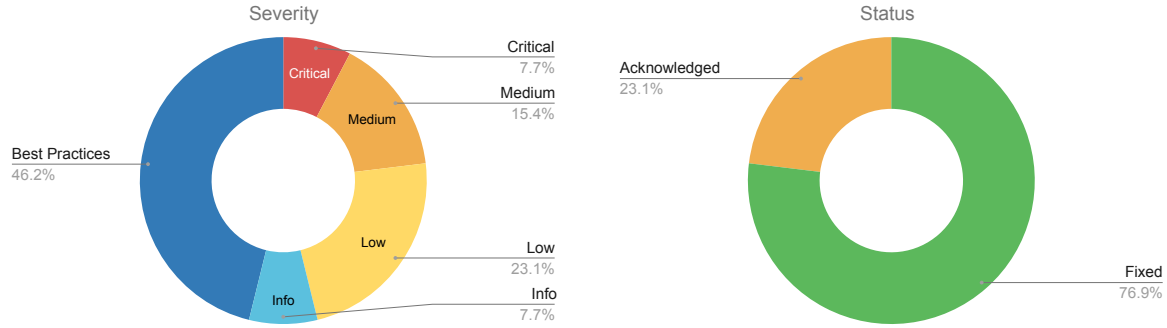
This document outlines the security review conducted by Nethermind Security for the Chi protocol contracts. Chi Protocol issued its first stablecoin, USC, using a dual stability mechanism to maintain its price at 1 USD. Users interact with the `ArbitrageV5` contract to mint and burn their USC tokens. This contract manages various scenarios, including fluctuations in USC price and the status of reserves, executing necessary actions to uphold the stability of USC's price, and ensuring adequate reserves that back the token. The protocol also includes mechanisms for rebalancing reserves within the `ReserveHolderV2` contract. It introduces specific adapter contracts like `WeEthAdapter` and `StEthAdapter` to manage the reserves of each supported LST.

The audited code comprises 1016 lines of code in Solidity. The **Chi Protocol** team has provided documentation that explains the purpose and functionality behind audited contracts.

At the time of our review, the code in the final commit hash reviewed by our team matches the code found in commit 3a4da8c4bc59a577a56fb551ed4bb7d657f235c7 of Chi Protocol's public repository. This note exclusively concerns the files within the defined scope, as listed in Section 2.

The audit was performed using: (a) manual analysis of the codebase, (b) simulation of the smart contracts. **Along this document, we report** 13 points of attention, where one is classified as `Critical`, two are classified as `Medium`, three are classified as `Low`, and seven are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

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



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (1), **High** (0), **Medium** (2), **Low** (3), **Undetermined** (0), **Informational** (1), **Best Practices** (6). **(b) Distribution of status: Fixed** (10), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jul 05, 2024
Final Report	Jul 22,2024
Methods	Manual Review, Automated analysis
Repository	chi-contracts
Commit Hash	5751cffb370de95dee9a7eb30bb1043985f5b8ae
Final Commit Hash	550c318d28736b29548609cc94beb43aae693f89
Documentation	Chi protocol documentation
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
01	contracts/ArbitrageV5.sol	531	43	8.1%	134	708
02	contracts/Treasury.sol	26	4	15.4%	8	38
03	contracts/library/ExternalContractAddresses.sol	15	2	13.3%	1	18
04	contracts/library/UniswapV3SwapLibrary.sol	52	1	1.9%	7	60
05	contracts/library/UniswapV2SwapLibrary.sol	47	1	2.1%	8	56
06	contracts/reserve/Adapter.sol	39	3	7.7%	8	50
07	contracts/reserve/ReserveHolderV2.sol	166	21	12.7%	43	230
08	contracts/reserve/StEthAdapter.sol	62	6	9.7%	16	84
09	contracts/reserve/WeEthAdapter.sol	78	6	7.7%	22	106
	Total	**1016**	**87**	**8.56%**	**247**	**1350**

3 Summary of Issues

	Finding	Severity	Update
1	Funds can be taken from the adapter due to missing access control in the `swapAmountToEth(...)` function	Critical	Fixed
2	Rescuing reserve tokens will result in DOS when claiming rewards and rebalancing other LSTs	Medium	Fixed
3	Wrong `discount` computation in the case of an excess of reserves	Medium	Fixed
4	Duplicated assets in the `reserveAssets` array lead to double accounting of reserves	Low	Fixed
5	Unimplemented `swapTolerance` functionality for rebalancing	Low	Fixed
6	Wrong accounting for `stETH` deposits will inflate the value of the reserves	Low	Fixed
7	Potential for incomplete configuration of reserve assets	Info	Acknowledged
8	External calls to arbitrary addresses in `ArbitrageV5::mint` and `ReserveHolderV2::deposit`	Best Practices	Fixed
9	Chainlink's price feeds are not using the `updatedAt` parameter	Best Practices	Acknowledged
10	Inconsistent usage of constants and the `asset` variable in adapter contracts	Best Practices	Fixed
11	Potential locked ETH in the adapter contracts	Best Practices	Acknowledged
12	Residual variables in storage due to contract upgrades	Best Practices	Fixed
13	The Removed adapter still have unlimited approval from the `ReserveHolderV2`	Best Practices	Fixed

4 System Overview

Chi protocol introduces their first stablecoin, USC, employing a dual stability mechanism designed to peg its value to 1 USD. This token is backed by reserves that include Liquid Staking Tokens (LSTs) and WETH.

The key contracts of the codebase include:

- **ArbitrageV5**: This primary contract implements USC's dual stability mechanism, ensuring its price remains stable at 1 USD and is fully backed by either LSTs or WETH.
- **ReserveHolderV2**: Responsible for managing the reserves backing USC in circulation, this contract holds WETH and interfaces with various adapters for LSTs.
- **StEthAdapter and WeEthAdapter**: Each Liquid Staking Token (LST) has its own adapter contract that holds and manages the respective LST reserves.

In subsequent sections, we delve into the detailed mechanics of Chi protocol's arbitrage mechanism and their approach to reserve management.

4.1 Arbitrage Mechanism

The `ArbitrageV5` contract manages the arbitrage operations, minting, and burning of USC tokens based on different scenarios.

We classify 6 distinct scenarios based on whether the USC price is above, below, or at the peg. Each scenario may result in either an excess or deficit of reserves. The `executeArbitrage(...)` function manages the logic for each scenario.

```
function executeArbitrage(uint256 maxChiSpotPrice) public override nonReentrant onlyArbitrager returns (uint256)
```

4.1.1 USC Price at 1 USD

When USC is valued at 1 USD, the contract evaluates the balance of reserves versus the circulating USC tokens:

- **Excess of Reserves:** If reserves exceed USC circulation. The `_arbitrageAtPegExcessOfReserves()` function is executed, minting a USC amount that is equivalent to the `reserveDiff`.

```
        function _arbitrageAtPegExcessOfReserves(uint256 reserveDiff, uint256 discount, uint256 ethPrice) private
        ↪  returns (uint256)
```

  The `discount` is calculated using the value of total USC in circulation to determine the portion of newly minted USC allocated to rewards.

```
        discount = (reserveDiff * 1 ether) / uscTotalSupplyValue;
```

- **Deficit of Reserves:** If reserves are insufficient relative to USC circulation, the `_arbitrageAtPegDeficitOfReserves()` function is executed. USC tokens are burned up to the reserve shortfall.

```
        function _arbitrageAtPegDeficitOfReserves(uint256 reserveDiff, uint256 discount, uint256 ethPrice)
        ↪  private returns (uint256)
```

  CHI tokens are minted and exchanged for WETH to augment reserves, with the minted USC including rewards based on the discount percentage.

```
        discount = (reserveDiff * 1 ether) / reserveValue;
```

4.1.2 USC Price Above 1 USD

When USC trades above 1 USD, the contract mints USC tokens to restore the peg:

- **Excess of Reserves:** The `_arbitrageAbovePegExcessOfReserves()` function verifies if the `reserveDiff` is sufficient to back the `delta`. In that case, the `deltaUSD` is swapped for CHI. Otherwise, the `reserveDiff` is swapped, with the remaining WETH amount (`deltaInETH` - `ethAmountToSwap`) sent to the reserves holder contract. The received CHI tokens are burned.

```
        function _arbitrageAbovePegExcessOfReserves(uint256 reserveDiff, uint256 ethPrice) private returns
        ↪  (uint256)
```

- **Deficit of Reserves:** When the contract has a deficit of reserves, the received WETH from the swap is sent to the reserves holder contract.

```
function _arbitrageAbovePegDeficitOfReserves(uint256 reserveDiff, uint256 ethPrice) private returns
↪ (uint256)
```

4.1.3 USC Price Below 1 USD

When USC is priced below 1 USD, the contract executes buyback operations using ETH:

- **Excess of Reserves:** Handled by `_arbitrageBelowPegExcessOfReserves()` function where surplus reserves cover the buyback of USC tokens, with excess USC burned.

```
function _arbitrageBelowPegExcessOfReserves(uint256 reserveDiff, uint256 ethPrice) private returns
↪ (uint256)
```

- **Deficit of Reserves:** The `_arbitrageBelowPegDeficitOfReserves()` function mints CHI tokens, swaps them for ETH, and subsequently for USC tokens. The purchased USC tokens are then burned to reduce the circulating supply.

```
function _arbitrageBelowPegDeficitOfReserves(uint256 reserveDiff, uint256 ethPrice) private returns
↪ (uint256)
```

4.2 Reserves Management

The `ReserveHolderV2` contract serves as the central repository for the Chi protocol's reserves, including WETH and various Liquid Staking Tokens (LSTs) managed through specific adapters.

- **Depositing Assets:**

  Users can deposit WETH or any supported LST into the reserves using the `deposit(...)` function. The deposited tokens are transferred from the caller to the contract and forwarded to the respective adapter.

```
function deposit(address reserveAsset, uint256 amount) external
```

- **Redeeming Assets:**

  The contract allows the Arbitrager contract to redeem WETH from the reserves using the `redeem(...)` function.

```
function redeem(uint256 amount) external onlyArbitrager returns (uint256)
```

- **Claiming Rewards:**

  Actors with the claimer role can claim rewards from all supported tokens using the `claimRewards(...)` function. This function iterates through each adapter, claims rewards, and then transfers them to the specified account.

```
function claimRewards(address account) external onlyClaimer
```

- **Rebalancing Reserves:**

  The contract implements a rebalancing mechanism through the `rebalance(...)` function, which adjusts reserves according to predefined percentages. It sells excessive tokens for LSTs with higher reserves than target and buys tokens with reserves lower than target.

```
function rebalance(uint256[] calldata protectionParams) external onlyRebalancer
```

Currently, Chi protocol supports two adapters: `WeEthAdapter` for managing WeETH and `StEthAdapter` for managing StETH tokens. Each adapter contract provides the following main functionalities:

- **Rescuing reserves:** The contract owner can withdraw all the contract's balance of the token through the `rescueReserves(...)` function.

```
function rescueReserves() external onlyOwner
```

- **Swap token for ETH:** The `swapAmountToEth(...)` function allows the exchange of a specific amount of tokens for ETH, and the received WETH is transferred to the specified receiver.

```
function swapAmountToEth(uint256 amountIn, uint256 minAmountOut, address receiver) external override returns
↪ (uint256)
```

- **Swap ETH for the token:** The `swapAmountFromEth(...)` function allows the exchange of an ETH amount for the adapter token; the WETH is transferred from the caller while the received tokens remain in the contract.

```
function swapAmountFromEth(uint256 amountIn) external returns (uint256)
```

4.3 USC Mint and Burn

Users interact with the `ArbitrageV5` contract to mint or burn USC tokens based on their needs.

- **Minting USC Tokens:**

  Users can mint USC tokens by depositing a specified token and amount using the `mint(...)` function. The equivalent amount of USC tokens is minted and transferred to the user.

```
function mint(address token, uint256 amount) external whenMintNotPaused nonReentrant
↪  onlyWhenMintableOrBurnable returns (uint256)
```

- **Burning USC Tokens:**

  Users can burn USC tokens to redeem the equivalent ETH amount from the reserves using the `burn(...)` function. The redeemed ETH is transferred back to the user.

```
function burn(uint256 amount) external whenBurnNotPaused nonReentrant onlyWhenMintableOrBurnable returns
↪  (uint256)
```

5 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 an attacker will uncover and exploit the finding. 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 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 formally pass to the client;

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] Funds can be taken from the adapter due to missing access control in the `swapAmountToEth(...)` function

File(s): `StEthAdapter.sol`, `WeEthAdapter.sol`

Description: The `swapAmountToEth(...)` function facilitates the exchange of the Adapter's asset to WETH, currently utilized by the `ReserveHolderV2` contract during its rebalancing process. This function swaps a specified `amountIn` of the asset and transfers the resulting WETH to the designated `receiver` address.

However, the current implementation lacks access control, allowing anyone to invoke the function, perform the swap, and transfer WETH to themselves, potentially depleting the contract's funds.

```
function swapAmountToEth(...) external override returns (uint256) {
  //@audit Lack of access control exposes the function to unauthorized use
  IERC20(stETH).approve(address(curvePool), amountIn);
  //@audit Tokens are swapped directly from the contract's balance
  uint256 ethReceived = curvePool.exchange(1, 0, amountIn, minAmountOut);

  totalDeposited -= amountIn;

  WETH.deposit{value: ethReceived}();
  //@audit Received WETH is transferred to the specified receiver
  IERC20(WETH).safeTransfer(receiver, ethReceived);

  return ethReceived;
}
```

Recommendation(s): Implement an access control mechanism to restrict the invocation of this function exclusively to the `ReserveHolderV2` contract.

Status: Fixed

Update from the client: 19222ab796faa6a3605fdc4dea92c636625efd18

6.2 [Medium] Rescuing reserve tokens will result in DOS when claiming rewards and rebalancing other LSTs

File(s): Adapter.sol, StEthAdapter.sol, WeEthAdapter.sol

Description: The Adapter provides rescueReserves(...) functionality for the owner to withdraw the LST assets in the reserves to the owner's address.

```
function rescueReserves() external onlyOwner {
   IERC20(asset).safeTransfer(msg.sender, IERC20(asset).balanceOf(address(this)));
   emit RescueReserves();
 }
```

However, as the assets are withdrawn but the totalDeposited will continue to hold the total deposits, calling the claimRewards(...) function will revert as the reward calculation will result in an underflow. Consequently, claiming rewards for all the LSTs reverts.

```
function claimRewards(address receiver) external onlyReserveHolder returns (uint256) {
    uint256 balance = IERC20(asset).balanceOf(address(this));
    uint256 reward = balance - totalDeposited;
    IERC20(asset).safeTransfer(receiver, reward);

    emit ClaimRewards(receiver, reward);
    return reward;
}
```

Similarly, the rebalance(...) function will revert once the swapAmountToEth(...) function is called in the adapter.

Recommendation(s): Consider skipping the adapter where funds are rescued when performing the rewards claiming or rebalancing process. This can be done by integrating a boolean flag that indicates if funds are rescued.

Status: Fixed

Update from the client: f7a0e724bbfc742ce32b9ffcb7291ab6b21c17ec

This function will be used in urgencies where we need to rescue reserves that are backing USC stablecoin in case of exploit or migration to new version of reserve holder. This means that funds will be moved in safe place and that arbitrage mechanism will also be changed manually in that case.

6.3 [Medium] Wrong discount computation in the case of an excess of reserves

File(s): ArbitrageV5.sol

Description: When the USC trades at peg (1 USD), there's an arbitrage opportunity through the excess or deficit of reserves. In that case, the discount is computed to represent the difference between the USC supply and reserves. The discount computation depends on the reserves status, as shown in the documentation :

```
if (isExcessOfReserves) {
    discount = (reserveDiff * 1 ether) / uscTotalSupplyValue;
} else{
    discount = (reserveDiff * 1 ether) / reserveValue;
}
```

However, the current implementation erroneously computes the discount in the case of an excess of reserves, as it divides the reserveDiff by the reserveValue instead of the uscTotalSupplyValue.

```
if (_almostEqualAbs(uscPrice, USC_TARGET_PRICE, pegPriceToleranceAbs)) {
    discount = Math.mulDiv(reserveDiff, BASE_PRICE, reserveValue);
}
```

Recommendation(s): Compute the discount based on the excess or deficit of reserves cases by dividing the reserveDiff by uscTotalSupplyValue or reserveValue, respectively.

Status: Fixed

Update from the client: e250fcc0f20d22aec0461f6b553e09378114b88a

6.4 [Low] Duplicated assets in the `reserveAssets` array lead to double accounting of reserves

File(s): ReserveHolderV2.sol

Description: The `reserveAssets` array stores the list of current reserve tokens; each token has its own adapter where funds are kept.

The contract owner can add a new reserve asset to the system using the `addReserveAsset(...)` function. However, this function currently lacks a check to prevent adding the same token multiple times.

```
function addReserveAsset(address reserveAsset) external onlyOwner {
  // @audit Possible duplicate asset in the array
  reserveAssets.push(reserveAsset);
  emit AddReserveAsset(reserveAsset);
}
```

When calculating the total reserves in the system, the `getReserveValue()` function iterates through the array of assets and sums their reserves. This results in duplicated token reserves being counted twice, which impacts the protocol's arbitrage functionality and breaks the system invariants.

```
function getReserveValue() public view returns (uint256) {
  // ...
  for (uint256 i = 0; i < reserveAssets.length; i++) {
    // @audit Reserve value is counted twice for duplicated tokens
    totalReserveValue += reserveAdapters[reserveAssets[i]].getReserveValue();
  }

  return totalReserveValue;
}
```

Recommendation(s): Ensure that the added token does not already exist in the array. This can be achieved by setting up the adapter with the same function and ensuring no previous adapter is set. Additionally, prevent setting the adapter address to `address(0)` in the `setReserveAssetAdapter(...)` function.

Status: Fixed

Update from the client: e7e87a14392da815e8d2719ac5526a8737f91527

6.5 [Low] Unimplemented `swapTolerance` functionality for rebalancing

File(s): ReserveHolderV2.sol

Description: `swapTolerance` is a configurable value that defines the tolerance window for performing sales and buys during the rebalancing process.

However, this variable is never initialized, and the contract does not provide any function to update its value. Consequently, `swapTolerance` will always default to `0` and does not impact the `maxReserveAssetValueForSell` and `minReserveAssetValueForBuy` computation.

```
function rebalance(...) external onlyRebalancer {
   // ...
   for (uint256 i = 0; i < reserveAssets.length; i++) {
      // ...
      uint256 reserveAssetTargetValue = Math.mulDiv(totalReserveValue, reserveAssetPercentage, MAX_PERCENTAGE);
      uint256 reserveAssetValue = reserveAdapters[reserveAsset].getReserveValue();
      uint256 maxReserveAssetValueForSell = Math.mulDiv(
        reserveAssetTargetValue,
        // @audit swapTolerance is always 0
        MAX_PERCENTAGE - swapTolerance,
        MAX_PERCENTAGE
      );
      // ...
      uint256 minReserveAssetValueForBuy = Math.mulDiv(
        reserveAssetTargetValue,
        // @audit swapTolerance is always 0
        MAX_PERCENTAGE + swapTolerance,
        MAX_PERCENTAGE
      );
  }
  // ...
}
```

Recommendation(s): Add an owner function to update the `swapTolerance` value.

Status: Fixed

Update from the client: 731d4649211dd22eeae33bfc9375c2e9eaef839b

6.6 [Low] Wrong accounting for `stETH` deposits will inflate the value of the reserves

File(s): `StEthAdapter.sol`

Description: There is a known issue with Lido's `stETH` token called the 1-2 wei corner case, which can be found in the official docs here. When a user transfers `stETH` to another user (this applies to deposit transactions, too), under the hood, the `stETH` balance gets converted to shares, integer division happens, and rounding down applies. This means the actual balance transferred is often off by 1-2 wei. You can read more about the math behind the issue here.

The `totalDeposited` will be off by 1-2 weis after each deposit, meaning that if a user deposits `1e18 stEth`, the actual amount deposited will be `0.99e18`.

This means that the calculation of the `ReserveHolderV2::getReserveValue` will be wrong and slightly inflated because the `StEthAdapter::getReserveValue` function uses `totalDeposited` when calculating the value of the reserves.

The `ArbitrageV5` contract relies on the precision of the reserves in the `_getReservesData` function, which will set the context for executing the arbitrage. In some cases, if the value of the `reserveValue` is very close to the value of `uscTotalSupplyValue`, the contract may assign the wrong value to the `isExcessOfReserves` boolean, and do wrong calculations.

Recommendation(s): Calculate the actual amount the user deposited, and update `totalDeposited` based on that, similarly to the flow of the `StEthAdapter::swapAmountFromEth` function.

Status: Fixed

Update from the client: c23ef6b45a870f7c51440c838de1b89bae14ce05

6.7 [Info] Potential for incomplete configuration of reserve assets

File(s): ReserveHolderV2.sol

Description: The `ReserveHolderV2` allows the Owner to add and support new LST tokens as reserves. However, the exposed functions to add support for new assets are segmented and could result in misconfigurations.

In order to support a new reserve asset, the owner will have to configure the below 3 steps.

1. Add a new Reserve Asset `addReserveAsset()` adds a new `reserveAsset` support in the `ReserveHolderV2`
2. Configure an adaptor for the asset `setReserveAssetAdapter()` adds an adapter for the new reserve asset
3. Add a percentage for the asset `setReserveAssetPercentage()` adds the percentage for the new reserve assets

If a new asset is added without configuring its adapter, the `getReserveValue(...)` function will revert, breaking any flow where it is used.

Similarly, if an asset's adapter is set but was not added to the array, deposits from the `ArbitrageV5` contract will go through (i.e calls to `mint(...)`) and funds are added to the right adapter. However, since the asset is not in the list, these reserves are not accounted for when calling the `getReserveValue()` function. Consequently, the collateral invariant is broken, as USC is minted, but the corresponding reserves are not accounted.

Recommendation(s): To avoid possible misconfigurations, we recommend setting up the asset and its adapter within the same function.

Status: Acknowledged

Update from the client: Partially resolved e7e87a14392da815e8d2719ac5526a8737f91527

Update from Nethermind Security: The provided commit does not full fix the issue. It is still possible to set the adapter for an inexistent asset. Consider ensuring that the token exists before setting the adapter within the `setReserveAssetAdapter()` function. This can be achieved by either iterating through the tokens array within the function, or using the `reserveAdapters` mapping.

6.8 [Best Practices] External calls to arbitrary addresses in `ArbitrageV5::mint` and `ReserveHolderV2::deposit`

File(s): `ArbitrageV5.sol`, `ReserveHolderV2`, `PriceFeedAggregator.sol`

Description: The current `ArbitrageV5::mint` function lacks validation for the `token` input parameter, allowing anyone to call `mint` with an arbitrary address. During execution, the contract invokes `safeTransferFrom` on the specified token address, enabling external calls and potentially allowing users to execute arbitrary logic. Similarly, the `deposit` function in `ReserveHolderV2` behaves similarly, accepting an `address reserveAsset` parameter and invoking `safeTransferFrom`.

While these arbitrary calls do not currently pose significant issues in the existing implementation, they could potentially impact external components interacting with the `ArbitrageV5` contract.

Furthermore, if a new Chainlink price feed is integrated into the price feed aggregator, a user could mint USC by providing the address of the corresponding token as an input parameter to the mint function. In this scenario, USC would be minted without the collateral being deposited into the adapters, thereby violating the protocol's 100% collateral invariant.

Recommendation(s): Revert the `ReserveHolderV2::deposit` transaction if a user attempts to deposit a token that is not supported and add input validation checks on the `token` address that the users pass inside the `ArbitrageV5::mint` function.

Status: Fixed

Update from the client: c23ef6b45a870f7c51440c838de1b89bae14ce05

6.9 [Best Practices] Chainlink's price feeds are not using the `updatedAt` parameter

File(s): ReserveHolderV2.sol

Description: During `rebalance`, the asset prices are read from the `priceFeeAggregator` which is used to compute the swap amount for the asset. The price is read from the aggregator using `peek()` function which sources the data from underlying oracles.

Below is a reference of how `reserveAssetAmountToSwap` is being computed using asset price.

```
uint256 reserveAssetAmountToSwap = Math.mulDiv(
        reserveAssetValueToSwap,
        10 ** IERC20Metadata(reserveAsset).decimals(),
        _peek(reserveAsset)
      );
```

The peek call is routed to the underlying oracle to fetch the price of the asset.

```
  function peek() external view returns (uint256 price) {
   (, int256 priceInUSD, , , ) = chainlinkFeed.latestRoundData();
   assert(priceInUSD > 0); // prices for assets from chainlink should always be grater than 0
   return uint256(priceInUSD);
  }
```

Chainlink provides the `updatedAt` parameter to track the last time the price was updated. This field allows the consumer protocol to recognise a stale price. Accurate price feed is critical to ensure USC holds the peg and does not depeg, as using stale prices without noticing will impact the valuation of USC stable coin.

Recommendation(s): Use the `updatedAt` value returned by the oracle. If the price is older than a configured threshold, a notification should be emitted for corrective action.

Status: Acknowledged

Update from the client:

6.10 [Best Practices] Inconsistent usage of constants and the `asset` variable in adapter contracts

File(s): `StEthAdapter.sol`, `WeEthAdapter.sol`

Description: The `StEthAdapter` and `WeEthAdapter` contracts utilize the `asset` variable alongside constants that store asset addresses: `stETH` and `weETH`. The current implementation does not enforce initializing `asset` to the specific constant, potentially leading to errors and complicating the code review process.

Recommendation(s): Consider initializing the `asset` variable with the appropriate token constant and consistently use it throughout the logic.

Status: Fixed

Update from the client: c2f609727e37870bcf13d33f4cdcac994b97a302

6.11 [Best Practices] Potential locked ETH in the adapter contracts

File(s): StEthAdapter.sol WeEthAdapter.sol

Description: The `StEthAdapter` contract implements a `receive()` function, allowing it to receive native Ether within the swap operations.

Within the `swapAmountToEth(...)` function, the contract receives native Ether when swapping the `stETH`, wrapping it into WETH. Likewise, within the `swapAmountFromEth(...)` function, transferred WETH to be swapped is unwrapped and received as native Ether in the contract.

However, any Ether that is received outside of these specific flows becomes locked within the contract, as there is no mechanism to withdraw it. Note that this applies to the `WeEthAdapter` contract as well.

Recommendation(s): In order to prevent locked Ether in the adapter contracts, restrict receiving Ether only from the necessary contracts.

Status: Acknowledged

Update from the client:

6.12 [Best Practices] Residual variables in storage due to contract upgrades

File(s): ReserveHolderV2.sol

description: The upgrade of `ReserveHolder` implementation contract resulted in the following variables not being used in the new implementation `ReserveHolderV2`. Their declaration is kept to retain the storage layout. However, the current code lacks comments or indicators that these variables are deprecated.

```
  uint256 public totalClaimed;
  uint256 public swapEthTolerance;
  uint256 public ethThreshold;
  uint256 public totalStEthDeposited;
  uint256 public curveStEthSafeGuardPercentage;
```

Recommendation(s): Add a comment against each of these variables as deprecated or rename them as deprecated.

Status: Fixed

Update from the client: New reserve holder contract will be deployed and funds will be migrated from old reserve holder.
c326f8e6720b4caf409a474afeba000e613ea48a

Update from Nethermind Security: The commit fixes the issue under the assumption that the contract will be re-deployed and not upgraded. Otherwise, removing the variable will break the existent storage layout of the contract.

6.13 [Best Practices] The Removed adapter still have unlimited approval from the `ReserveHolderV2`

File(s): `ReserveHolderV2.sol`

Description: When a new adapter is assigned to a specific asset using the `setReserveAssetAdapter(...)` function, it receives unlimited approval for both the reserve asset and WETH. This setup facilitates the execution of deposit and swap functionalities.

However, if the asset adapter is updated to a new address, the unlimited approval granted to the previous address persists. While this does not pose a significant risk currently, it could potentially be exploited in future implementations of the `ReserveHolderV2` contract.

```
function setReserveAssetAdapter(address reserveAsset, address adapter) external onlyOwner {
  //@audit Approval for the previous adapter persists
  reserveAdapters[reserveAsset] = IAdapter(adapter);

  IERC20(reserveAsset).approve(adapter, type(uint256).max);
  IERC20(WETH).approve(adapter, type(uint256).max);

  emit SetReserveAssetAdapter(reserveAsset, adapter);
}
```

Recommendation(s): When updating to a new adapter address, consider resetting the approval for the previous adapter to `0`.

Status: Fixed

Update from the client: 550c318d28736b29548609cc94beb43aae693f89

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 Chi Protocol documentation

The **Chi Protocol** team has provided documentation about their protocol through their official documentation which includes a general overview of the protocol as well as a detailed description of the arbitrage process and its different flows. The information in the documentation is concise and technical, with well-written explanations.
Additionally, the **Chi Protocol** team was available to address any questions or concerns from the Nethermind Security team.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge build
[] Compiling...
[] Compiling 179 files with 0.8.25
[] Solc 0.8.25 finished in 21.82s
Compiler run successful with warnings:
Warning (5740): Unreachable code.
   --> node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol:116:9:
    |
116 |         return true;
    |         ^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol:162:9:
    |
162 |         return true;
    |         ^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol:228:9:
    |
228 |         uint256 fromBalance = _balances[from];
    |         ^ (Relevant source part starts here and spans across multiple lines).

Warning (5740): Unreachable code.
   --> node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol:230:9:
    |
230 |         unchecked {
    |         ^ (Relevant source part starts here and spans across multiple lines).

Warning (5740): Unreachable code.
   --> node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol:256:9:
    |
256 |         _totalSupply += amount;
    |         ^ (Relevant source part starts here and spans across multiple lines).

Warning (5740): Unreachable code.
   --> node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol:282:9:
    |
282 |         uint256 accountBalance = _balances[account];
    |         ^ (Relevant source part starts here and spans across multiple lines).

Warning (5740): Unreachable code.
   --> node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol:284:9:
    |
284 |         unchecked {
    |         ^ (Relevant source part starts here and spans across multiple lines).
```

8.2 Tests Output

8.2.1 Arbitrage tests

```
> forge test --match-path "test/arbitrage/*"

Ran 40 tests for test/arbitrage/Arbitrage.t.sol:ArbitrageTest
[PASS] testFuzz_Burn(uint256) (runs: 259, : 186381, ~: 186474)
[PASS] testFuzz_CalculateDeltaETH(uint256) (runs: 256, : 136404, ~: 136397)
[PASS] testFuzz_CalculateDeltaUSC(uint256) (runs: 256, : 145560, ~: 145552)
[PASS] testFuzz_ExecuteArbitrage_PrivilegedSpotPriceTooBig(uint256) (runs: 259, : 399947, ~: 400259)
[PASS] testFuzz_ExecuteArbitrage_RevertChiSpotPriceTooBig(uint256) (runs: 259, : 43663, ~: 43944)
[PASS] testFuzz_Mint(uint256,uint256,uint256,uint256) (runs: 259, : 283944, ~: 281086)
[PASS] testFuzz_Mint_RevertReserveDiffTooBig(uint256,uint256,uint256) (runs: 259, : 187309, ~: 187311)
[PASS] testFuzz_Mint_RevertUscPriceNotPegged(uint256,uint256,uint256,bool) (runs: 259, : 165401, ~: 164869)
[PASS] testFuzz_SetChiPriceTolerance(uint16) (runs: 259, : 20155, ~: 19990)
[PASS] testFuzz_SetChiPriceTolerance_RevertNotOwner(address,uint16) (runs: 259, : 12189, ~: 12189)
[PASS] testFuzz_SetChiPriceTolerance_RevertToleranceTooBig(uint16) (runs: 256, : 12647, ~: 12647)
[PASS] testFuzz_SetMaxMintBurnPriceDiff(uint256) (runs: 259, : 14839, ~: 14932)
[PASS] testFuzz_SetMaxMintBurnPriceDiff_RevertNotOwner(address,uint256) (runs: 259, : 12203, ~: 12203)
[PASS] testFuzz_SetMaxMintBurnReserveTolerance(uint16) (runs: 259, : 20050, ~: 19864)
[PASS] testFuzz_SetMaxMintBurnReserveTolerance_RevertNotOwner(address,uint16) (runs: 259, : 12197, ~: 12197)
[PASS] testFuzz_SetMaxMintBurnReserveTolerance_RevertToleranceTooBig(uint16) (runs: 256, : 12691, ~: 12691)
[PASS] testFuzz_SetMintBurnFee(uint16) (runs: 259, : 19665, ~: 19786)
[PASS] testFuzz_SetMintBurnFee_RevertFeeTooBig(uint16) (runs: 256, : 12667, ~: 12667)
[PASS] testFuzz_SetMintBurnFee_RevertNotOwner(address,uint16) (runs: 259, : 12195, ~: 12195)
[PASS] testFuzz_SetPegPriceToleranceAbs(uint256) (runs: 259, : 30690, ~: 30844)
[PASS] testFuzz_SetPegPriceToleranceAbs_RevertNotOwner(address,uint256) (runs: 259, : 12093, ~: 12093)
[PASS] testFuzz_SetPriceTolerance(uint16) (runs: 259, : 20133, ~: 19912)
[PASS] testFuzz_SetPriceTolerance_RevertNotOwner(address,uint16) (runs: 259, : 12243, ~: 12243)
[PASS] testFuzz_SetPriceTolerance_RevertToleranceTooBig(uint16) (runs: 256, : 12724, ~: 12724)
[PASS] testFuzz_UpdateArbitrager(address,bool) (runs: 259, : 23750, ~: 33124)
[PASS] testFuzz_UpdateArbitrager_RevertNotOwner(address,address,bool) (runs: 259, : 12462, ~: 12462)
[PASS] testFuzz_UpdatePrivileged(address,bool) (runs: 259, : 23106, ~: 13272)
[PASS] testFuzz_UpdatePrivileged_RevertNotOwner(address,address,bool) (runs: 259, : 12552, ~: 12552)
[PASS] test_ExecuteArbitrage_Arbitrage1_BigDelta() (gas: 371869)
[PASS] test_ExecuteArbitrage_Arbitrage2() (gas: 296647)
[PASS] test_ExecuteArbitrage_Arbitrage3_BigDelta() (gas: 374992)
[PASS] test_ExecuteArbitrage_Arbitrage3_SmallDelta() (gas: 366477)
[PASS] test_ExecuteArbitrage_Arbitrage4_BigDelta() (gas: 382388)
[PASS] test_ExecuteArbitrage_Arbitrage4_SmallDelta() (gas: 372067)
[PASS] test_ExecuteArbitrage_Arbitrage5() (gas: 241763)
[PASS] test_ExecuteArbitrage_Arbitrage6() (gas: 242203)
[PASS] test_ExecuteArbitrage_RevertChiPriceNotPegged(uint16) (runs: 259, : 132045, ~: 131847)
[PASS] test_ExecuteAribtrage_Arbitrage1_SmallDelta() (gas: 366717)
[PASS] test_MintPause() (gas: 24327)
[PASS] test_SetBurnPause() (gas: 13887)
Suite result: ok. 40 passed; 0 failed; 0 skipped; finished in 272.64s (2064.39s CPU time)
```

8.2.2 Tests output

```
> forge test

Ran 9 tests for test/staking/ChiStaking.t.sol:ChiStakingTest
[PASS] test_Lock_DontUseStakedTokens() (gas: 305879)
[PASS] test_SetUp() (gas: 12995)
[PASS] test_Unstake_MultipleStakers() (gas: 567199)
[PASS] test_Unstake_MultipleStakers_UnstakeFromChiLocking() (gas: 554264)
[PASS] test_Unstake_OnlyOneStakerWithRewards() (gas: 381634)
[PASS] test_Unstake_OnlyOneStakerWithoutRewards() (gas: 285082)
[PASS] test_UpdateEpoch_WithStakersWithRewards() (gas: 583699)
[PASS] test_UpdateEpoch_WithStakersWithoutRewards() (gas: 394135)
[PASS] test_UpdateEpoch_WithoutStakersWithoutRewards() (gas: 52256)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 34.16ms (9.92ms CPU time)

Ran 18 tests for test/staking/USCStaking.t.sol:USCStakingTest
[PASS] testFuzz_SetRewardController(address) (runs: 259, : 31157, ~: 31157)
[PASS] testFuzz_SetRewardController_Revert_NotOwner(address,address) (runs: 259, : 12123, ~: 12123)
[PASS] test_ClaimStEth() (gas: 1205882)
[PASS] test_ClaimStEth_StakeBetweenClaim() (gas: 1379223)
[PASS] test_ClaimStEth_WithoutRewards() (gas: 742304)
[PASS] test_ClaimUSCRewards() (gas: 1223092)
[PASS] test_SetUp() (gas: 26366)
[PASS] test_Stake_MultipleStakers_MultipleEpochs() (gas: 583871)
[PASS] test_Stake_MultipleStakers_MultipleEpochs_WithRewards() (gas: 1413267)
[PASS] test_Stake_OnlyOneStaker_MultipleEpochs_WithoutRewards() (gas: 382816)
[PASS] test_Stake_OnlyOneStaker_OnlyOneEpoch() (gas: 229948)
[PASS] test_Stake_Revert_ZeroAmount() (gas: 8388)
[PASS] test_Unstake_MultipleStakers() (gas: 796787)
[PASS] test_Unstake_OnlyOneStakerWithRewards() (gas: 489056)
[PASS] test_Unstake_OnlyOneStakerWithoutRewards() (gas: 293133)
[PASS] test_UpdateEpoch_WithStakersWithRewards() (gas: 773978)
[PASS] test_UpdateEpoch_WithStakersWithoutRewards() (gas: 429778)
[PASS] test_UpdateEpoch_WithoutStakersWithoutRewards() (gas: 70424)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 122.41ms (97.15ms CPU time)

Ran 6 tests for test/oracle/PriceFeedAggregator.t.sol:PriceFeedAggregatorTest
[FAIL. Reason: unknown cheatcode with selector 0x59e02dd7; you may have a mismatch between the `Vm` interface (likely in
↪  `forge-std`) and the `forge` version; counterexample:
↪  calldata=0x27ac117a0000000000000000000000000000000000000000000000000000000000000000fa7626d400000000000000000000000007109709
ecfa91a80626ff3989d68f67f5b1dd12d0000000000000000000000000000000000000000000000000000000000000000000000000000000019730000000000000000
0000000000000000000000000000000000000000000000000000000000000025 args=[0x0000000000000000000000000000000000fA7626D4,
↪  0x7109709ECfa91a80626fF3989D68f67F5b1DD12D, 6515, 37]] testFuzz_Peek(address,address,uint256,uint8) (runs: 2, :
↪  39787, ~: 39787)
[PASS] testFuzz_Peek_Revert_ZeroAddress() (gas: 10714)
[PASS] testFuzz_SetPriceFeed(address,address) (runs: 259, : 36427, ~: 36427)
[PASS] testFuzz_SetPriceFeed_RevertZeroAddress(address) (runs: 259, : 18423, ~: 18423)
[PASS] testFuzz_SetPriceFeed_Revert_NonOwner(address) (runs: 259, : 12039, ~: 12039)
[PASS] test_SetUp() (gas: 7727)
Suite result: FAILED. 5 passed; 1 failed; 0 skipped; finished in 134.82ms (110.51ms CPU time)

Ran 6 tests for test/sale/TimeWeightedBonding.t.sol:TimeWeightedBondingTest
[PASS] testFuzz_RecoverChi(uint256) (runs: 259, : 38302, ~: 38066)
[PASS] testFuzz_RecoverChi_Revert_NotOwner(address,uint256) (runs: 259, : 11953, ~: 11953)
[PASS] testFuzz_Vest_Revert_NotOwner(address,address,uint256) (runs: 259, : 12255, ~: 12255)
[PASS] test_Buy() (gas: 238744)
[PASS] test_SetUp() (gas: 19930)
[PASS] test_Vest() (gas: 96660)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 177.46ms (151.65ms CPU time)

Ran 12 tests for test/staking/ChiVesting.t.sol:ChiVestingTest
[PASS] testFuzz_SetRewardController_Revert_NotOwner(address,address) (runs: 259, : 12234, ~: 12234)
[PASS] testFuzz_SetVester(address,bool) (runs: 259, : 23697, ~: 33148)
[PASS] testFuzz_SetVester_Revert_NotOwner(address,address,bool) (runs: 259, : 12459, ~: 12459)
[PASS] testFuzz_setRewardController(address) (runs: 259, : 32308, ~: 32308)
[PASS] test_AddVesting() (gas: 528078)
[PASS] test_AvailableChi() (gas: 914548)
[PASS] test_ClaimStEth() (gas: 844464)
[PASS] test_GetVotingPower() (gas: 696366)
[PASS] test_SetUp() (gas: 22713)
[PASS] test_UnclaimedStETHAmount() (gas: 823856)
```

```
[PASS] test_UpdateEpoch() (gas: 658296)
[PASS] test_WithdrawChi() (gas: 982705)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 185.31ms (160.14ms CPU time)

Ran 8 tests for test/tokens/USC.t.sol:USCTest
[PASS] testFuzz_Burn(address,uint256) (runs: 259, : 71992, ~: 72119)
[PASS] testFuzz_Burn_Revert_NotMinter(address,uint256) (runs: 259, : 12092, ~: 12092)
[PASS] testFuzz_Mint(address,uint256) (runs: 259, : 88575, ~: 89037)
[PASS] testFuzz_Mint_Revert_NotMinter(address,uint256) (runs: 259, : 11462, ~: 11462)
[PASS] testFuzz_UpdateMinter(address,bool) (runs: 259, : 27473, ~: 18100)
[PASS] testFuzz_UpdateMinter_Revert_NotOwner(address,address,bool) (runs: 259, : 12334, ~: 12334)
[PASS] testFuzz_UpdateMinter_Revert_ZeroAddress(bool) (runs: 259, : 10945, ~: 10945)
[PASS] test_SetUp() (gas: 10094)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 115.27ms (263.16ms CPU time)

Ran 10 tests for test/tokens/CHI.t.sol:USCTest
[PASS] testFuzz_Burn(address,uint256) (runs: 259, : 58464, ~: 58471)
[PASS] testFuzz_BurnFrom(address,address,uint256) (runs: 259, : 79559, ~: 79993)
[PASS] testFuzz_BurnFrom_Revert_NotMinter(address,address,uint256) (runs: 259, : 13088, ~: 13088)
[PASS] testFuzz_Burn_Revert_NotMinter(address,uint256) (runs: 259, : 12125, ~: 12125)
[PASS] testFuzz_Mint(address,uint256) (runs: 259, : 71689, ~: 71937)
[PASS] testFuzz_Mint_Revert_NotMinter(address,uint256) (runs: 259, : 11484, ~: 11484)
[PASS] testFuzz_UpdateMinter(address,bool) (runs: 259, : 27111, ~: 18122)
[PASS] testFuzz_UpdateMinter_Revert_NotOwner(address,address,bool) (runs: 259, : 12390, ~: 12390)
[PASS] testFuzz_UpdateMinter_Revert_ZeroAddress(bool) (runs: 259, : 10967, ~: 10967)
[PASS] test_SetUp() (gas: 16184)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 299.00ms (448.41ms CPU time)

Ran 14 tests for test/staking/ChiLocking.t.sol:ChiLockingTest
[PASS] testFuzz_SetChiLocker(address,bool) (runs: 259, : 22405, ~: 13185)
[PASS] testFuzz_SetChiLocker_Revert_NotOwner(address,address,bool) (runs: 259, : 12387, ~: 12387)
[PASS] testFuzz_SetRewardController(address) (runs: 259, : 32222, ~: 32222)
[PASS] testFuzz_SetRewardController_Revert_NotOwner(address,address) (runs: 259, : 12119, ~: 12119)
[PASS] testFuzz_SetUscStaking(address) (runs: 259, : 32511, ~: 32511)
[PASS] testFuzz_SetUscStaking_Revert_NotOwner(address,address) (runs: 259, : 12195, ~: 12195)
[PASS] test_AvailableChiWithdraw() (gas: 1345699)
[PASS] test_ClaimStEth() (gas: 1590314)
[FAIL. Reason: assertion failed] test_GetVotingPower() (gas: 1318887)
[PASS] test_LockChi() (gas: 1256463)
[PASS] test_SetUp() (gas: 16144)
[PASS] test_UnclaimedStEthAmount() (gas: 1554120)
[PASS] test_UpdateEpoch() (gas: 1513865)
[PASS] test_WithdrawChiFromAccount() (gas: 1597579)
Suite result: FAILED. 13 passed; 1 failed; 0 skipped; finished in 290.70ms (247.86ms CPU time)

Ran 5 tests for test/oracle/ChainlinkOracle.t.sol:ChainlinkOracleTest
[PASS] testFork_Decimals() (gas: 27827)
[PASS] testFork_Name() (gas: 44251)
[PASS] testFork_Peek() (gas: 55978)
[PASS] testFork_SetUp() (gas: 12898)
[PASS] testFuzz_Peek(uint256) (runs: 259, : 10651, ~: 10651)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 4.88s (1.61s CPU time)

Ran 8 tests for test/oracle/UniswapV2TwapOracle.t.sol:UniswaV2TwapOracleTest
[PASS] testFork_Decimals() (gas: 5531)
[PASS] testFork_Name() (gas: 13737)
[PASS] testFork_Peek() (gas: 117795)
[PASS] testFork_SetUp() (gas: 42137)
[PASS] testFork_getTwapQuote_MinPeriodNotPassed() (gas: 226449)
[PASS] testFork_getTwapQuote_MinPeriodPassed() (gas: 226365)
[PASS] testFork_updateCumulativePricesSnapshot() (gas: 90653)
[PASS] testFork_updateCumulativePricesSnapshot_Revert_PeriodNotPassed() (gas: 25908)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 6.97s (4.69ms CPU time)

Ran 17 tests for test/dso/OCHI.t.sol:oChiTest
[PASS] testFork_Burn() (gas: 1740544)
[PASS] testFork_Burn_Revert_NotApprovedOrOwner() (gas: 643818)
[PASS] testFork_Burn_Revert_OptionExpired() (gas: 2791267)
[PASS] testFork_Burn_Revert_OptionLocked() (gas: 1772817)
[PASS] testFork_ClaimRewards_OnlyHisRewards() (gas: 1335512)
```

```
[PASS] testFork_ClaimRewards_Revert_NotAllowed() (gas: 643874)
[PASS] testFork_GetUnclaimedRewardsValue() (gas: 1296538)
[FAIL. Reason: assertion failed] testFork_Mint_BothLPTokensWithChi() (gas: 693958)
[FAIL. Reason: assertion failed] testFork_Mint_BothLPTokensWithoutChi() (gas: 681993)
[PASS] testFork_Mint_DiscountCap() (gas: 665280)
[PASS] testFork_Mint_Revert_InvalidLockPeriod() (gas: 15561)
[PASS] testFork_Mint_Revert_PolTargetRatioExceeded() (gas: 71167)
[FAIL. Reason: assertion failed] testFork_Mint_WithoutLPTokensWithChi() (gas: 404004)
[PASS] testFork_RecoverLPTokens() (gas: 1029100)
[PASS] testFork_UpdateEpoch() (gas: 175483)
[PASS] testFork_UpdateEpoch_Revert_EpochNotFinished() (gas: 23674)
[PASS] test_SetUp() (gas: 66801)
Suite result: FAILED. 14 passed; 3 failed; 0 skipped; finished in 182.56s (4.08s CPU time)

Ran 20 tests for test/reserve-holder/ReserveHolderV2.t.sol:ReserveHolderTest
[PASS] testForkFuzz_AddReserveAsset(address) (runs: 259, : 62016, ~: 62016)
[PASS] testForkFuzz_AddReserveAsset_Revert_NotOwner(address,address) (runs: 259, : 13514, ~: 13514)
[PASS] testForkFuzz_ClaimRewards(uint256,uint256) (runs: 259, : 248129, ~: 248223)
[PASS] testForkFuzz_Deposit_WETH(uint256) (runs: 259, : 98427, ~: 98529)
[PASS] testForkFuzz_Deposit_stETH(uint256) (runs: 259, : 298082, ~: 298123)
[PASS] testForkFuzz_Deposit_weETH(uint256) (runs: 259, : 266291, ~: 266384)
[FAIL. Reason: assertion failed] testForkFuzz_Rebalance() (gas: 1044983)
[PASS] testForkFuzz_Redeem(uint256,uint256) (runs: 259, : 238766, ~: 238773)
[PASS] testForkFuzz_Redeem_Revert_NotArbitrager(uint256) (runs: 259, : 10751, ~: 10751)
[PASS] testForkFuzz_SetArbitrager(address,bool) (runs: 259, : 30855, ~: 40383)
[PASS] testForkFuzz_SetArbitrager_Revert_NotOwner(address,address,bool) (runs: 259, : 13749, ~: 13749)
[PASS] testForkFuzz_SetClaimer(address) (runs: 259, : 39683, ~: 39683)
[PASS] testForkFuzz_SetClaimer_Revert_NotOwner(address,address) (runs: 259, : 13592, ~: 13592)
[PASS] testForkFuzz_SetRebalancer(address,bool) (runs: 259, : 30440, ~: 20606)
[PASS] testForkFuzz_SetRebalancer_Revert_NotOwner(address,address,bool) (runs: 259, : 13801, ~: 13801)
[FAIL. Reason: EvmError: Revert; counterexample:
↪  calldata=0x5d89854c0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000 args=[0x0000000000000000000000000000000000000000,
↪  0x0000000000000000000000000000000000000000]] testForkFuzz_SetReserveAssetAdapter(address,address) (runs: 0, : 0, ~:
↪  0)
[PASS] testForkFuzz_SetReserveAssetAdapter_Revert_NotOwner(address,address,address) (runs: 259, : 13847, ~: 13847)
[PASS] testForkFuzz_SetReserveAssetPercentage(address,uint256) (runs: 259, : 44639, ~: 44791)
[PASS] testForkFuzz_SetReserveAssetPercentage_Revert_NotOwner(address,address,uint256) (runs: 259, : 13673, ~: 13673)
[PASS] testForkFuzz_SetReserveAssetPercentage_Revert_PercentageTooHigh(address,uint256) (runs: 259, : 20835, ~: 20727)
Suite result: FAILED. 18 passed; 2 failed; 0 skipped; finished in 263.33s (299.82s CPU time)

Ran 4 tests for test/reserve-holder/WeEthAdapter.t.sol:WeEthAdapterTest
[PASS] testForFuzz_SwapExactAmountFromEth(uint256,uint256) (runs: 259, : 374338, ~: 374260)
[PASS] testForFuzz_SwapExactAmountInWeEth(uint256,uint256) (runs: 259, : 412482, ~: 236449)
[PASS] testForkFuzz_ClaimRewards(uint256,uint256) (runs: 259, : 161436, ~: 161327)
[PASS] testForkFuzz_Deposit(uint256) (runs: 259, : 119027, ~: 119100)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 323.24s (204.79s CPU time)

Ran 17 tests for test/staking/RewardController.t.sol:RewardControllerTest
[PASS] testFuzz_SetArbitrager(address) (runs: 259, : 32354, ~: 32354)
[PASS] testFuzz_SetArbitrager_Revert_NotOwner(address,address) (runs: 259, : 11866, ~: 11866)
[PASS] testFuzz_SetChiIncentivesForChiEthLPStaking(uint256) (runs: 259, : 30741, ~: 30741)
[PASS] testFuzz_SetChiIncentivesForChiEthLPStaking_Revert_NotOwner(address,uint256) (runs: 259, : 11686, ~: 11686)
[PASS] testFuzz_SetChiIncentivesForChiLocking(uint256) (runs: 259, : 30730, ~: 30807)
[PASS] testFuzz_SetChiIncentivesForChiLocking_Revert_NotOwner(address,uint256) (runs: 259, : 11816, ~: 11816)
[PASS] testFuzz_SetChiIncentivesForUscEthLPStaking(uint256) (runs: 259, : 30327, ~: 30789)
[PASS] testFuzz_SetChiIncentivesForUscEthLPStaking_Revert_NotOwner(address,uint256) (runs: 259, : 11708, ~: 11708)
[PASS] testFuzz_SetChiIncentivesForUscStaking(uint256) (runs: 259, : 30700, ~: 30854)
[PASS] testFuzz_SetChiIncentivesForUscStaking_Revert_NotOwner(address,uint256) (runs: 259, : 11678, ~: 11678)
[PASS] testFuzz_UnclaimedStETHAmount(address) (runs: 259, : 44279, ~: 44279)
[PASS] test_ClaimStEth() (gas: 155826)
[PASS] test_RewardUSC() (gas: 122760)
[PASS] test_RewardUSC_Revert_NotArbitrager() (gas: 12757)
[PASS] test_RewardUSC_Revert_ZeroAmount() (gas: 54381)
[FAIL. Reason: StEthYieldPercentagesNotCorrect()] test_UpdateEpoch() (gas: 273064)
[PASS] test_UpdateEpoch_Revert_EpochNotFinished() (gas: 13307)
Suite result: FAILED. 16 passed; 1 failed; 0 skipped; finished in 323.42s (450.21ms CPU time)
```

```
Ran 5 tests for test/reserve-holder/StEthAdapter.t.sol:StEthAdapterTest
[PASS] testForFuzz_SwapAmountToEth(uint256,uint256,address) (runs: 259, : 256515, ~: 256046)
[PASS] testForFuzz_SwapExactAmountInEth(uint256,uint256) (runs: 259, : 300334, ~: 300371)
[PASS] testForkFuzz_ClaimRewards(uint256,uint256,address) (runs: 259, : 187970, ~: 188017)
[PASS] testForkFuzz_Deposit(uint256) (runs: 259, : 163086, ~: 163189)
[PASS] testForkFuzz_RevertNotReserveHolder() (gas: 130846)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 359.84s (628.06s CPU time)

Ran 17 tests for test/reserve-holder/ReserveHolder.t.sol:ReserveHolderTest
[PASS] testForkFuzz_Deposit(uint256) (runs: 259, : 153274, ~: 153967)
[PASS] testForkFuzz_Deposit_Revert_TransferFailed(uint256) (runs: 259, : 46411, ~: 46411)
[PASS] testForkFuzz_Rebalance_EthAboveThreshold(uint256) (runs: 259, : 249826, ~: 249866)
[PASS] testForkFuzz_Rebalance_EthBellowThreshold(uint256) (runs: 259, : 335821, ~: 335849)
[PASS] testForkFuzz_RedeemWithSwap(uint256,uint256) (runs: 259, : 360841, ~: 361801)
[PASS] testForkFuzz_RedeemWithoutSwap(uint256,uint256) (runs: 259, : 116258, ~: 116917)
[PASS] testForkFuzz_Redeem_Revert_NotArbitrager(address,uint256) (runs: 259, : 13011, ~: 13011)
[PASS] testForkFuzz_SetArbitrager(address,bool) (runs: 259, : 31610, ~: 40370)
[PASS] testForkFuzz_SetArbitrager_Revert_NotOwner(address,address,bool) (runs: 259, : 13726, ~: 13726)
[PASS] testForkFuzz_SetClaimer(address) (runs: 259, : 22508, ~: 22527)
[PASS] testForkFuzz_SetClaimer_Revert_NotOwner(address,address) (runs: 259, : 13515, ~: 13515)
[PASS] testForkFuzz_SetEthThreshold(uint256) (runs: 259, : 26522, ~: 26730)
[PASS] testForkFuzz_SetEthThreshold_Revert_NotOwner(address,uint256) (runs: 259, : 13345, ~: 13345)
[PASS] testForkFuzz_SetEthThreshold_Revert_ThresholdTooHigh(uint256) (runs: 259, : 20586, ~: 20409)
[PASS] testForkFuzz_SetSwapEthTolerance(uint256) (runs: 259, : 21879, ~: 21935)
[PASS] testForkFuzz_SetSwapEthTolerance_Revert_NotOwner(address,uint256) (runs: 259, : 13500, ~: 13500)
[PASS] testFork_SetUp() (gas: 23248)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 372.24s (394.68s CPU time)

Ran 13 tests for test/dso/LPRewards.t.sol:LPRewardsTest
[PASS] testFork_LockLP() (gas: 108123)
[PASS] testFork_LockLP_Revert_NotOCHI() (gas: 12680)
[PASS] testFork_RecoverLPTokens_Revert_NotOCHI() (gas: 12748)
[PASS] testFork_SetOCHI_Revert_NotOwner() (gas: 12835)
[PASS] testFork_SetUp() (gas: 20548)
[PASS] testFork_UpdateEpoch() (gas: 844200)
[PASS] testFuzzFork_LockLP(uint256,int256,uint64) (runs: 259, : 115630, ~: 115615)
[PASS] testFuzzFork_LockLP_Revert_TokenIdAlreadyUsed(uint256,int256,uint256,uint64) (runs: 259, : 158289, ~: 158344)
[PASS] testFuzzFork_RecoverLPTokens(uint256) (runs: 259, : 43099, ~: 43084)
[PASS] testFuzzFork_SetOCHI(address) (runs: 259, : 15184, ~: 15184)
[PASS] test_CalculateUnclaimedRewards() (gas: 618970)
[PASS] test_ClaimRewards_OnlyHisRewards() (gas: 472743)
[PASS] test_ClaimRewards_RewardsFromExpired() (gas: 510820)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 372.24s (2.75s CPU time)

Ran 18 test suites in 372.36s (2582.41s CPU time): 221 tests passed, 8 failed, 0 skipped (229 total tests)
```

Remarks about Chi Protocol's test suite

The development team behind **Chi Protocol** has a comprehensive fuzz testing suite that checks the main functionalities of the protocol such as the different arbitrage scenarios, the mint and burn flows.
However, note that the test suite is based on the **ArbitrageV3** contract instead of **ArbitrageV5**. Furthermore, the **testForkFuzzRebalance** test fails because of the 1-2 wei corner case issue with **stETH**.
To ensure robust testing coverage, we recommend updating the test suite to align with the V5 version of the Arbitrage contract and addressing the specific issues to ensure all tests pass successfully. This will help maintain the integrity and reliability of the protocol's functionalities under various scenarios.

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 procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, 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 is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our 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.

Learn more about us at nethermind.io.

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.