Security Review Report
NM-0556 Permissionless Technologies USPD Contracts

NETHERMIND
SECURITY

(August 20, 2025)

Contents

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for the Permissionless Technologies' **USPD Stablecoin protocol** contracts.

USPD is an ERC20-compliant USD-pegged stablecoin that maintains its peg through an NFT-based overcollateralization mechanism. The protocol implements a dual-token architecture: **cUSPDToken** (core non-rebasing shares) handles minting/burning logic and collateral management, while **USPDToken** provides a rebasing view layer for user-friendly 1:1 USD representation.

The system uses **Stabilizer NFTs** for priority-based collateral allocation via a linked list structure and **Position NFTs** for tracking individual backing positions. A liquidation mechanism with variable thresholds based on stabilizer NFT IDs enables automatic stabilizer takeover. Supporting infrastructure includes **PriceOracle** for ETH/USD pricing, **OvercollateralizationReporter** for monitoring, escrow contracts for fund management, **PoolSharesConversionRate** for yield factor conversion, and **BridgeEscrow** for cross-chain functionality.

This audit focused on the ten core smart contracts: **UspdToken**, **cUSPDToken**, **StabilizerNFT**, **PositionEscrow**, **StabilizerEscrow**, **PriceOracle**, **OvercollateralizationReporter**, **BridgeEscrow**, **InsuranceEscrow**, and **PoolSharesConversionRate**, along with their interfaces and the complete NFT-based collateralization system.

The audit comprises 2043 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 12 points of attention, where three are classified as `Critical`, two are classified as `High`, two are classified as `Medium`, two are classified as `Low` and three 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.

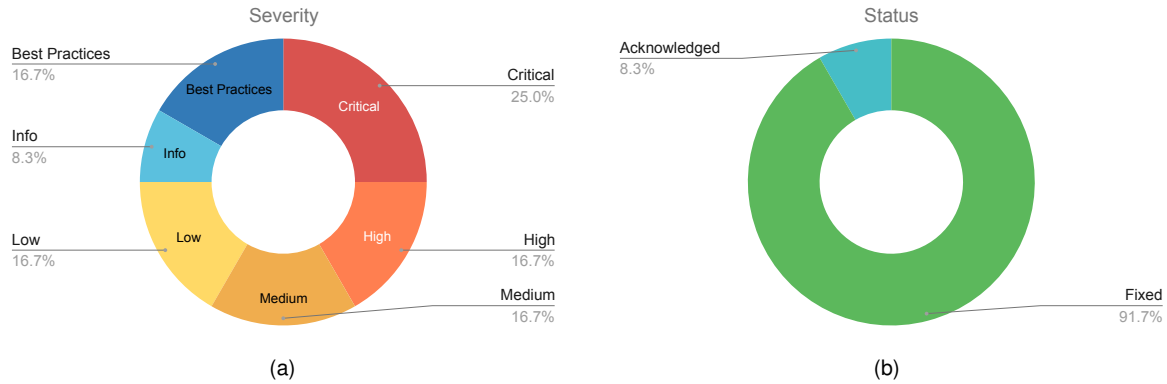

Fig. 1: Distribution of issues: Critical (3), **High** (2), **Medium** (2), **Low** (2), **Undetermined** (0), **Informational** (1), **Best Practices** (2). **Distribution of status: Fixed** (11), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Final Report	August 20, 2025
Initial Commit Hash	`f3e3bccd05d3c3a3aa73285ef31f00e27a36866f1`
Final Commit Hash	`a148cd614e5f13d7c78397b5a093e12b00d067bc`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	BridgeEscrow.sol	123	54	43.9%	29	206
2	PositionEscrow.sol	168	118	70.2%	59	345
3	PriceOracle.sol	271	63	23.2%	55	389
4	PoolSharesConversionRate.sol	54	72	133.3%	22	148
5	UspdToken.sol	172	119	69.2%	39	330
6	cUSPDToken.sol	145	109	75.2%	38	292
7	InsuranceEscrow.sol	34	32	94.1%	9	75
8	StabilizerNFT.sol	647	307	47.4%	187	1141
9	OvercollateralizationReporter.sol	135	103	76.3%	40	278
10	StabilizerEscrow.sol	70	66	94.3%	26	162
11	interfaces/IStabilizerNFT.sol	18	13	72.2%	6	37
12	interfaces/IPositionEscrow.sol	33	13	39.4%	6	52
13	interfaces/IBridgeEscrow.sol	32	22	68.8%	5	59
14	interfaces/ICreateX.sol	25	45	180.0%	8	78
15	interfaces/IPriceOracle.sol	16	1	6.2%	3	20
16	interfaces/IPoolSharesConversionRate.sol	9	35	388.9%	6	50
17	interfaces/IStabilizerEscrow.sol	15	13	86.7%	6	34
18	interfaces/IInsuranceEscrow.sol	14	25	178.6%	7	46
19	interfaces/IcUSPDToken.sol	35	36	102.9%	9	80
20	interfaces/ILido.sol	4	11	275.0%	1	16
21	interfaces/IOvercollateralizationReporter.sol	23	39	169.6%	10	72
	Total	**2043**	**1296**	**63.4%**	**571**	**3910**

3 Summary of Issues

	Finding	Severity	Update
1	Stabilizer owners can avoid being collateralized by manipulating the global collateralization ratio	Critical	Fixed
2	Uniswap ETH/USDC price is incorrectly fetched, causing price deviation checks to always revert	Critical	Fixed
3	`MINIMUM_UNALLOCATE_COLLATERALIZATION_RATIO` is incorrectly set, rendering the system collateralization ratio check ineffective	Critical	Fixed
4	Transferring the Stabilizer NFT does not transfer the `EXCESSCOLLATERALMANAGER_ROLE` in the `PositionEscrow` contract	High	Fixed
5	Unallocated shares from `unallocateStabilizerFunds` are not refunded to the user and are permanently lost	High	Fixed
6	Malicious Stabilizers can permanently DoS the protocol	Medium	Fixed
7	Stabilizers Can Set a Minimum Collateralization Ratio That Makes Their Positions Immediately Liquidatable	Medium	Fixed
8	Misconfigured Signer Can Cause Denial of Service in `attestationService(...)`	Low	Fixed
9	Stale yield factor from queued messages can devalue L2 USPD	Low	Acknowledged
10	`setLiquidationParameters` should enforce `_payoutPercent` to be strictly greater than `100`	Info	Fixed
11	Address is used before being validated	Best Practices	Fixed
12	The shortfall condition in `liquidatePosition` is always true and can safely be removed	Best Practices	Fixed

4 System Overview

USPD Protocol is a yield-bearing, overcollateralized stablecoin system built on Ethereum that enables users to mint USD-pegged tokens backed by **stETH** (Lido Staked Ethereum) collateral. The protocol features a dual-token architecture comprising **USPD** (rebasing token for users) and cUSPD (non-rebasing shares for DeFi integrations), with collateralization managed through NFT-based positions. The system's design facilitates cross-chain deployments, automated yield distribution, and price oracle integration. It is worth mentioning that the protocol design keeps track of NFT positions as a linked list and iterates over it on every allocation or unallocation, in which it could lead to Gas Exhaustion during the execution of transactions. The client acknowledged and is aware of this.

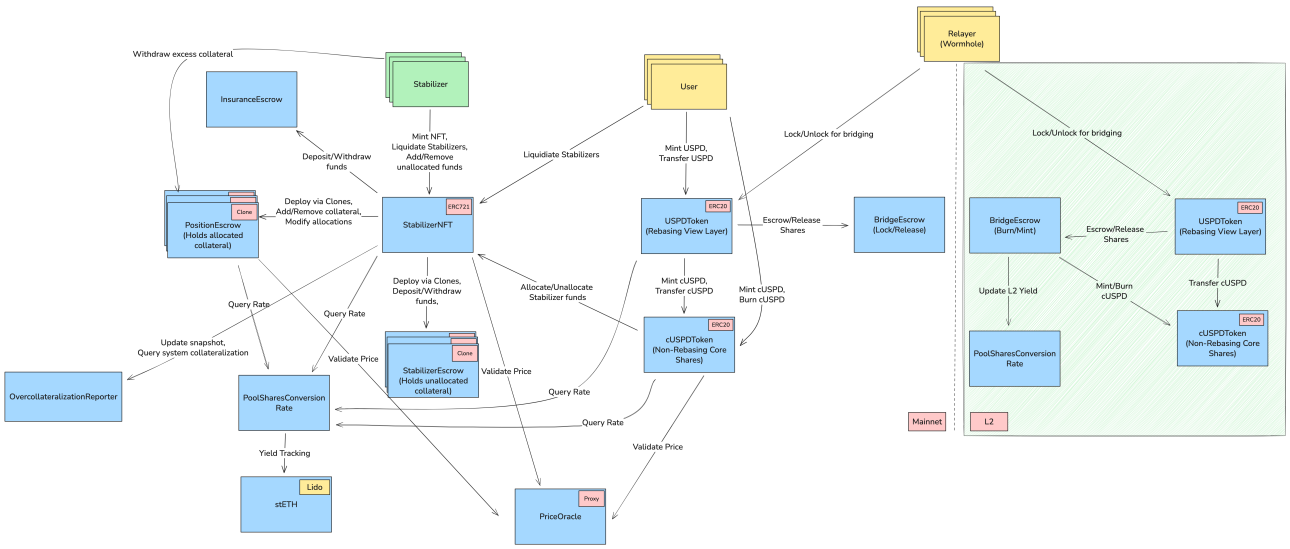

Fig. 2: USPD Protocol Overview

4.1 Actors

- **Users**: Primary participants who interact with the protocol by minting and burning **cUSPD** tokens. Users receive rebasing **USPD** tokens that automatically reflect yield accrual from the underlying **stETH** collateral.
- **Stabilizers**: Entities that provide **stETH** collateral to the system to ensure overcollateralization. These providers enable the minting of **cUSPD** tokens and earn fees for maintaining healthy collateralization ratios. Each stabilizer position is represented by a **StabilizerNFT**.
- **Liquidators**: Actors who monitor for undercollateralized stabilizer positions and initiate liquidations to maintain system solvency, earning a reward in the process.
- **Admins**: Protocol governance entities responsible for system configuration, upgrades, and emergency controls. Admins manage the multi-source price oracle system, configure cross-chain bridge parameters, execute contract upgrades via UUPS proxies.
- **Bridge Operators**: Entities that facilitate cross-chain operations by managing the locking and unlocking of **cUSPD** shares in the **BridgeEscrow** contract. These operators enable **USPD** to be deployed on Layer 2 networks while maintaining backing collateral on Ethereum mainnet.
- **Oracle Signers**: Off-chain actors running the **USPD** price oracle, which provides a real-time ETH/USD price. Their signed price data is validated against on-chain sources including Chainlink feeds and WETH/USDC Uniswap pools, with automatic invalidation if deviations exceed 5

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] Stabilizer owners can avoid being collateralized by manipulating the global collateralization ratio

File(s): /src/PositionEscrow.sol

Description: For each Stabilizer NFT, a dedicated `PositionEscrow` contract is deployed to manage the associated collateral. The protocol continuously tracks the global collateralization ratio as a core safety metric, updating it during collateral allocations and removals (e.g., within `allocateStabilizerFunds`, `unallocateStabilizerFunds`, and `liquidatePosition`). For example, in `allocateStabilizerFunds`:

```
function allocateStabilizerFunds(
    uint256 ethUsdPrice,
    uint256 priceDecimals
) external payable override returns (AllocationResult memory result) {
    // --SNIP
    // --- Update Snapshot via Reporter ---
    if (result.totalEthEquivalentAdded > 0) {
        reporter.updateSnapshot(int256(result.totalEthEquivalentAdded));
    }
    // --SNIP
}
```

The `PositionEscrow` contract tracks locked collateral using the ERC20 `balanceOf` method. Collateral additions are intended to be made through the functions `PositionEscrow::addCollateralEth` and `PositionEscrow::addCollateralStETH`, both of which correctly notify the system of new collateral:

```
function addCollateralStETH(uint256 stETHAmount) external override /* Removed role check */ {
    if (stETHAmount == 0) revert ZeroAmount();

    // Pull stETH from the caller
    bool success = IERC20(stETH).transferFrom(msg.sender, address(this), stETHAmount);
    if (!success) revert TransferFailed(); // Check allowance and balance

    // Emit event acknowledging the stETH added to the pool
    emit CollateralAdded(stETHAmount);

    // ethValue = stEth * yield
    // Report addition to StabilizerNFT
    IStabilizerNFT(stabilizerNFTContract).reportCollateralAddition(stETHAmount);
}
```

Stabilizer owners can bypass these reporting functions by transferring `stETH` directly to their `PositionEscrow` via a standard ERC20 transfer. Since `PositionEscrow` tracks its collateral balance using the ERC20 `balanceOf` method, this directly increases the escrow's perceived collateral without triggering any collateral addition report. As a result, the global collateralization ratio becomes underreported.

This underreporting can be exploited in several ways:

1) Consider the following example:

1.a) There are two positions with the following allocations:

- 1st Position: backed cUSPD = $100 | collateral value = $90 | collateralRatio = 90%
- 2nd Position: backed cUSPD = $100 | collateral value = $90 | collateralRatio = 90%

Global system collateralization ratio is 90%

1.b) Both positions can be liquidated, the 1st stabilizer directly transfers stETH to his Position Escrow to increase his collateral value to $110. This position's allocation is updated as follows:

- 1st Position: backed cUSPD = $100 | collateral value = $110 | collateralRatio = 110%
- 2nd Position: backed cUSPD = $100 | collateral value = $90 | collateralRatio = 90%

Crucially, the global system collateralization ratio is still 90% since the last collateral increase was not reported.

1.c) Both positions can still be liquidated. A liquidator submits a TX to liquidate the 1st position. However, given that the system is under-reporting its collateral ratio (90%), the liquidate transaction will revert in the following line, assuming that the position's collateral is healthier than the system, while it is not:

```
    function liquidatePosition(
        uint256 liquidatorTokenId,
        uint256 positionTokenId,
        uint256 cuspdSharesToLiquidate,
        IPriceOracle.PriceAttestationQuery calldata priceQuery
    ) external {
        // --SNIP
         if (address(reporter) != address(0)) {
            uint256 systemCollateralRatio = reporter.getSystemCollateralizationRatio(priceResponse);

            if (systemCollateralRatio != type(uint256).max
                && positionCollateralRatio > systemCollateralRatio) {
                revert LiquidationNotBelowSystemRatio();
            }
        }
    }
```

Here, the `positionCollateralRatio` is `110%` while the `systemCollateralRatio` is `90%`.

2) Moreover, after directly funding their escrow, a stabilizer can call `removeExcessCollateral` to withdraw the unreported collateral. The system will report the removal, further desynchronizing the global collateral state, since the addition was never reported.

```
    function removeExcessCollateral(
        address payable recipient,
        uint256 amountToRemove,
        IPriceOracle.PriceAttestationQuery calldata priceQuery
    ) external override onlyRole(EXCESSCOLLATERALMANAGER_ROLE) {
        // --SNIP
        // Report removal to StabilizerNFT
        IStabilizerNFT(stabilizerNFTContract).reportCollateralRemoval(amountToRemove);
    }
```

These issues can lead to a denial of service for other users, prevent legitimate liquidations, and severely undermine the protocol's risk management and collateral accounting.

Recommendation(s): Consider tracking the stETH locked in EscrowPositions via internal state, this forces stabilizer owners to pass through `addCollateralEth` and `addCollateralStETH` where the collateral addition is reported.

Status: Fixed

Update from the client: Fixed in commits a0319f4, 9afc63f and 64b074d

6.2 [Critical] Uniswap ETH/USDC price is incorrectly fetched, causing price deviation checks to always revert

File(s): /src/PriceOracle.sol

Description: The `PriceOracle::attestationService` function is used throughout the system to calculate the price of ETH. This function relies on a price attestation from the Morpho oracle and verifies the price against Chainlink and Uniswap as a deviation check:

```
function attestationService(
    PriceAttestationQuery calldata priceQuery
) public payable whenNotPaused returns (PriceResponse memory) {
    // --SNIP
    if (block.chainid == 1) {
        // Get prices from other sources
        uint256 chainlinkPrice = uint256(
            getChainlinkDataFeedLatestAnswer()
        );
        if (chainlinkPrice == 0) {
            revert PriceSourceUnavailable("Chainlink");
        }

        uint256 uniswapV3Price = getUniswapV3WethUsdcPrice();
        if (uniswapV3Price == 0) {
            revert PriceSourceUnavailable("Uniswap V3");
        }

        // Check price deviations
        if (!_isPriceDeviationAcceptable(priceQuery.price,chainlinkPrice,uniswapV3Price)) {
            revert PriceDeviationTooHigh(priceQuery.price,chainlinkPrice,uniswapV3Price);
        }
    }
    // --SNIP
}
```

The issue arises in the `getUniswapV3WethUsdcPrice` function:

```
function getUniswapV3WethUsdcPrice() public view returns (uint) {
    IUniswapV3Factory factory = IUniswapV3Factory(uniswapV3Factory);
    address uniswapV3PoolWethUSDC = factory.getPool(
        uniswapRouter.WETH(),
        usdcAddress,
        3000
    );
    if (uniswapV3PoolWethUSDC != address(0)) {
        IUniswapV3PoolState uniswapPoolState = IUniswapV3PoolState(uniswapV3PoolWethUSDC);
        (uint sqrtPriceX96, , , , , , ) = uniswapPoolState.slot0();
==>         return ((sqrtPriceX96 / 2 ** 96) ** 2) * 1e12;
    }

    return 0;
}
```

The function incorrectly assumes that `sqrtPriceX96` from slot0 gives the price of ETH in terms of USDC, when in reality, for USDC/WETH pool, it gives the price of USDC in terms of ETH (because, in the USDC/WETH pool, USDC is token0). As a result, the returned price is inverted and significantly larger than the actual ETH price in USDC. This leads to the price deviation check always reverting, effectively causing a permanent denial of service to the protocol.

Recommendation(s): Consider correcting the implementation to invert the price so it accurately reflects the price of ETH in terms of USDC.

Status: Fixed

Update from the client: Fixed in commit 3e9fce8

6.3 [Critical] `MINIMUM_UNALLOCATE_COLLATERALIZATION_RATIO` is incorrectly set, rendering the system collateralization ratio check ineffective

File(s): `/src/StabilizerNFT.sol`

Description: When burning cUSPD, the `unallocateStabilizerFunds` function is called to withdraw the corresponding collateral. Before proceeding, the function checks that the global system collateralization ratio is healthy (i.e., above 100%):

```
function unallocateStabilizerFunds(
    uint256 poolSharesToUnallocate, // Changed parameter name
    IPriceOracle.PriceResponse memory priceResponse
) external override returns (uint256 unallocatedEth) {
    // --SNIP
    if (address(reporter) != address(0)) {
        uint256 currentSystemRatio = reporter.getSystemCollateralizationRatio(priceResponse);
        if (currentSystemRatio < MINIMUM_UNALLOCATE_COLLATERALIZATION_RATIO) {
            revert SystemUnstableUnallocationNotAllowed();
        }
    }
}
```

However, there is a critical misalignment between the scaling of the value returned by `reporter.getSystemCollateralizationRatio` and the value of `MINIMUM_UNALLOCATE_COLLATERALIZATION_RATIO`. The former is scaled by `10000`, while the latter is not:

```
uint256 public constant MINIMUM_UNALLOCATE_COLLATERALIZATION_RATIO = 100; // e.g., 100 for 100%
```

As a result, the current value of 100 actually represents a 1% collateralization ratio when compared against the scaled `currentSystemRatio`, effectively making the collateralization check meaningless.

Recommendation(s): Consider updating the value of `MINIMUM_UNALLOCATE_COLLATERALIZATION_RATIO` with the scaling used in the return value of `getSystemCollateralizationRatio`.

Status: Fixed

Update from the client: Fixed in commit 862e3d8

6.4 [High] Transferring the Stabilizer NFT does not transfer the EXCESSCOLLATERALMANAGER_ROLE in the PositionEscrow contract

File(s): /src/StabilizerNFT.sol

Description: When a user mints a stabilizer NFT, two contracts are initialized: `PositionEscrow` and `StabilizerEscrow`. During initialization, the `EXCESSCOLLATERALMANAGER_ROLE`—which permits the holder to withdraw excess collateral via `removeExcessCollateral`—is granted to the stabilizer NFT owner:

```
function initialize( // <-- Renamed from constructor
    address _stabilizerNFT,
    address _stabilizerOwner,
    address _stETHAddress,
    address _lidoAddress,
    address _rateContractAddress,
    address _oracleAddress
) external initializer { // <-- Added initializer modifier
    __AccessControl_init(); // <-- Initialize AccessControl

    if (_stabilizerNFT == address(0) || _stabilizerOwner == address(0) || _stETHAddress == address(0) || _lidoAddress ==
    ↪  address(0) || _rateContractAddress == address(0) || _oracleAddress == address(0)) {
        revert ZeroAddress();
    }

    stabilizerNFTContract = _stabilizerNFT;
    stETH = _stETHAddress;
    lido = _lidoAddress;
    rateContract = _rateContractAddress;
    oracle = _oracleAddress;

    _grantRole(DEFAULT_ADMIN_ROLE, _stabilizerNFT);
    _grantRole(STABILIZER_ROLE, _stabilizerNFT);
    _grantRole(EXCESSCOLLATERALMANAGER_ROLE, _stabilizerOwner);
}
```

However, when the stabilizer NFT is transferred to a new account, the `EXCESSCOLLATERALMANAGER_ROLE` is not revoked from the previous owner nor granted to the new owner. The `_update` function, which is invoked on transfers, does not handle role updates:

```
// StabilizerNFT.sol

function _update(address to, uint256 tokenId, address auth)
    internal
    override(ERC721Upgradeable, ERC721EnumerableUpgradeable)
    returns (address)
{
    return super._update(to, tokenId, auth);
}
```

This allows the previous owner to continue withdrawing excess collateral, while the new owner does not gain the necessary permissions.

Recommendation(s): Consider revoking the `EXCESSCOLLATERALMANAGER_ROLE` role from the old owner and grant it to the new one upon NFT transfer.

Status: Fixed

Update from the client: Fixed in commits 7335fab and f20f5bf

6.5 [High] Unallocated shares from `unallocateStabilizerFunds` are not refunded to the user and are permanently lost

File(s): /src/StabilizerNFT.sol

Description: When users burn their cUSPD tokens by calling `cUSPDToken::burnShares`, the function immediately burns the specified number of shares and then calls `StabilizerNFT::unallocateStabilizerFunds` to unallocate the corresponding collateral and refund it to the user:

```
function burnShares(
    uint256 sharesAmount,
    address payable to,
    IPriceOracle.PriceAttestationQuery calldata priceQuery
) external returns (uint256 unallocatedStEthReturned) {
    // --SNIP
==>     _burn(msg.sender, sharesAmount);

    // 3. Unallocate funds via StabilizerNFT
    uint256 unallocatedStEth = stabilizer.unallocateStabilizerFunds(
        sharesAmount,
        oracleResponse
    );
    // --SNIP

}
```

Within `unallocateStabilizerFunds`, the function iterates backward over the allocated positions to withdraw the equivalent ETH of the burned shares. Notably, the implementation enforces a minimum remaining gas requirement before continuing the loop, which is intended to prevent DoS condition in cases with a large number of allocated positions:

```
function unallocateStabilizerFunds(
    uint256 poolSharesToUnallocate, // Changed parameter name
    IPriceOracle.PriceResponse memory priceResponse
) external override returns (uint256 unallocatedEth) {
    // --SNIP
    uint256 currentId = highestAllocatedId;
    uint256 remainingPoolShares = poolSharesToUnallocate;
    uint256 totalUserStEthReturned; //default = 0;
    uint256 totalEthEquivalentRemovedAggregate; //default = 0;

    while (currentId != 0 && remainingPoolShares > 0) {
        if (gasleft() < MIN_GAS) break;
        // --SNIP
    }
     require(totalUserStEthReturned > 0, "No funds unallocated");

    if (totalEthEquivalentRemovedAggregate > 0) {
        reporter.updateSnapshot(-int256(totalEthEquivalentRemovedAggregate));
    }

    return totalUserStEthReturned;
}
```

If the loop exits early due to gas exhaustion, the function may not fully unallocate all of the shares passed in `poolSharesToUnallocate`. However, since the entire amount is already burned in `cUSPDToken::burnShares` before calling `unallocateStabilizerFunds`, any unprocessed shares are neither refunded to the user nor retained, resulting in a permanent loss of those shares.

Recommendation(s): Consider modifying the logic to burn only the shares that have been successfully unallocated in the StabilizerNFT

Status: Fixed

Update from the client: Fixed in commit a811587

6.6 [Medium] Malicious Stabilizers can permanently DoS the protocol

File(s): /src/StabilizerNFT.sol

Description: When minting cUSPD, the token contract calls `StabilizerNFT::allocateStabilizerFunds` to over-collateralize the minted tokens. The function iterates over unallocated positions to back the new tokens:

```
function allocateStabilizerFunds(
    // poolSharesToMint removed
    uint256 ethUsdPrice,
    uint256 priceDecimals
) external payable override returns (AllocationResult memory result) {
    require(msg.sender == address(cuspdToken), "Only cUSPD contract"); // Check against cUSPD
    require(lowestUnallocatedId != 0, "No unallocated funds");
    require(msg.value > 0, "No ETH sent"); // User must send ETH

    uint256 currentId = lowestUnallocatedId;
    uint256 remainingEth = msg.value;
    while (currentId != 0 && remainingEth > 0) {
        // --SNIP
    }
}
```

For each unallocated position, the function determines the required collateral, withdraws it from the `StabilizerEscrow`, and attempts to deposit it into the `PositionEscrow`:

```
uint256 stabilizerStEthNeeded = (remainingEth * (pos.minCollateralRatio - 10000)) / 10000;
// Determine how much stabilizer stETH can actually be allocated (min of needed and available)
uint256 toAllocate = stabilizerStEthNeeded > escrowBalance
    ? escrowBalance
    : stabilizerStEthNeeded;

// If stabilizer can't provide the ideally needed amount, adjust the user's ETH share accordingly
uint256 userEthShare = remainingEth;
if (toAllocate < stabilizerStEthNeeded) {
    userEthShare = (toAllocate * 10000) / (pos.minCollateralRatio - 10000);

    if (userEthShare > remainingEth) {
        userEthShare = remainingEth;
    }
}

// --- Interact with PositionEscrow ---
// PositionNFT interaction removed
address positionEscrowAddress = positionEscrows[currentId];

// 1. Transfer Stabilizer's stETH from StabilizerEscrow to PositionEscrow
// Approve this contract to pull from StabilizerEscrow
IStabilizerEscrow(escrowAddress).approveAllocation(toAllocate, address(this));
// Pull the funds
bool successStabilizer = IERC20(stETH).transferFrom(escrowAddress, positionEscrowAddress, toAllocate);
if (!successStabilizer) revert("Stabilizer stETH transfer to PositionEscrow failed");

// 2. Call PositionEscrow.addCollateralFromStabilizer
// This sends the user's ETH (userEthShare) which gets converted to stETH inside PositionEscrow,
// and acknowledges the stabilizer's stETH (toAllocate) that we just transferred.
IPositionEscrow(positionEscrowAddress).addCollateralFromStabilizer{value: userEthShare}(toAllocate);
```

The `addCollateralFromStabilizer` function in `PositionEscrow` stakes the ETH into Lido and reverts if either staking fails or the returned stETH is zero:

```
function addCollateralFromStabilizer(uint256 stabilizerStEthAmount)
    external
    payable
    override
    onlyRole(STABILIZER_ROLE)
{
    uint256 userEthAmount = msg.value;
    if (userEthAmount == 0 && stabilizerStEthAmount == 0) revert ZeroAmount(); // Must add something

    uint256 userStEthReceived; //default 0
    if (userEthAmount > 0) {
        // Stake User's ETH via Lido - stETH is minted directly to this contract
        try ILido(lido).submit{value: userEthAmount}(address(0)) returns (uint256 receivedStEth) {
            userStEthReceived = receivedStEth;
==>             if (userStEthReceived == 0) revert TransferFailed(); // Lido submit should return > 0 stETH
        } catch {
            revert TransferFailed(); // Lido submit failed
        }
    }

    // --SNIP
}
```

Since `userEthShare` is ultimately determined by the available collateral in a stabilizer's position (`min(unallocated funds, msg.value)`:

```
// StabilizerNFT::allocateStabilizerFunds

uint256 toAllocate = stabilizerStEthNeeded > escrowBalance
    ? escrowBalance
    : stabilizerStEthNeeded;
```

a malicious stabilizer can create a position with an extremely small amount of collateral (e.g., 1 wei). During allocation, this minimal value will be submitted to Lido for staking. Due to the internal Lido logic, staking such a small amount will return zero stETH, causing `addCollateralFromStabilizer` to revert and thus reverting the entire mint transaction:

```
// Lido.sol

function getSharesByPooledEth(uint256 _ethAmount) public view returns (uint256) {
    return _ethAmount
        .mul(_getTotalShares())
        .div(_getTotalPooledEther());
}
```

At the time of writing, the value of `_getTotalShares` and `_getTotalPooledEther` are `753647327920189655481114 8` and `910724450461297259-544963`, respectively. So, staking 1 wei would result in zero stETH due to Lido's proportional share calculation.

Because the allocation logic always processes positions starting from the lowest unallocated position and there is no privileged mechanism to remove or skip malicious positions, a single malicious position can indefinitely block the minting of new cUSPD tokens, permanently DoSing to the protocol.

Recommendation(s): Consider enforcing a minimum collateral amount for stabilizer positions

Status: Fixed

Update from the client: Fixed in commits 531e688..a4ce22b

6.7 [Medium] Stabilizers Can Set a Minimum Collateralization Ratio That Makes Their Positions Immediately Liquidatable

File(s): /src/StabilizerNFT.sol

Description: The `StabilizerNFT` contract allows a stabilizer to set a minimum collateralization ratio (`minCollateralRatio`) for their position using the `setMinCollateralizationRatio(...)` function. This ratio determines the collateral level when their funds are allocated via `allocateStabilizerFunds(...)`. Another feature, `liquidatePosition(...)`, allows users to liquidate positions that are under a specific collateralization threshold.

The problem is a logical flaw where the minimum allowed `minCollateralRatio` is lower than the liquidation thresholds available to lower NFT Id holders. A stabilizer can set their `minCollateralRatio` to `11000` (110%), which is the lowest permitted value.

```
function setMinCollateralizationRatio(
    uint256 tokenId,
    uint256 newRatio
) external {
    require(ownerOf(tokenId) == msg.sender, "Not token owner");
    // @audit-issue A user can set a ratio of 11000 (110%).
    require(newRatio >= 11000 && newRatio <= 100000, "Ratio must be at between 110.00% and 1000%");
    // ...
    positions[tokenId].minCollateralRatio = newRatio;
    // ...
}
```

However, the liquidation system gives an advantage to holders of low-numbered NFTs. The owner of `liquidatorTokenId 1` can liquidate any position with a collateral ratio below `12500` (125%), and this threshold decreases for higher token IDs.

```
function liquidatePosition(...) external {
    // ...
    uint256 calculatedThreshold;
    if (liquidatorTokenId == 0) {
        calculatedThreshold = 11000;
    } else {
        // ...
        if (((liquidatorTokenId - 1) * 50) >= (12500 - 11000)) {
             calculatedThreshold = 11000;
        } else {
            // @audit The threshold can be as high as 12500 for liquidatorTokenId 1.
            calculatedThreshold = 12500 - ((liquidatorTokenId - 1) * 50);
        }
    }

    // @audit A position with a ratio of 11000 will pass this check if the
    // liquidator has a privileged token ID (e.g., ID 1-30).
    require(
        positionCollateralRatio < calculatedThreshold,
        "Position not below liquidation threshold"
    );
    // ...
}
```

This creates a scenario where a stabilizer setting their minimum collateralization ratio to 110% will find their position immediately liquidatable by any NFT holder with an ID between `1` and `30`, because their set minimum (110%) is below the liquidation thresholds for those NFT IDs (e.g., ID 30's threshold is 110.5%). These positions become available for immediate liquidation, despite the stabilizer's intention to set a reasonable minimum.

Recommendation(s): Consider increasing the minimum value for `newRatio` in the `setMinCollateralizationRatio(...)` function. The minimum should be greater than the highest possible liquidation threshold.

Status: Fixed

Update from the client: Fixed in commits f14708a

6.8 [Low] Misconfigured Signer Can Cause Denial of Service in `attestationService(...)`

File(s): `/src/PriceOracle.sol`

Description: The `attestationService(...)` function in the `PriceOracle` contract is called by other protocol contracts to verify signed price data supplied by end-users. The function uses the `priceQuery.dataTimestamp` field to prevent two types of issues: data being too old (staleness) and old data being replayed after newer data has already been submitted.

The function correctly checks that the provided `dataTimestamp` is not too far in the past relative to `block.timestamp`. However, it lacks a corresponding check to ensure the `dataTimestamp` is not unreasonably far in the future.

```
function attestationService(
    PriceAttestationQuery calldata priceQuery
) public payable whenNotPaused returns (PriceResponse memory) {
    // ...

    // @audit A misconfigured signer can set `lastAttestationTimestamp` to a large future value,
    // causing this check to fail for all subsequent legitimate calls.
    if ((priceQuery.dataTimestamp + 1000 * 15) < lastAttestationTimestamp) {
        revert StaleAttestation(
            lastAttestationTimestamp,
            priceQuery.dataTimestamp
        );
    }

    // ...

    // @audit This update allows a misconfigured signer to poison the contract's state
    // with a large future timestamp, breaking the replay protection.
    lastAttestationTimestamp = priceQuery.dataTimestamp;
    // ...
}
```

This could lead to a DoS if a signer is misconfigured. For instance, a bug in the off-chain signing software or an incorrect system clock could cause a signer to submit an attestation with a large future timestamp. If this transaction is processed, it would set `lastAttestationTimestamp` to this incorrect future value. Consequently, all subsequent attestations from correctly configured signers would fail the replay protection check causing them to revert. This would unintentionally halt all price updates and disrupt protocol operations that rely on the oracle

Recommendation(s): Consider adding an upper-bound check on `priceQuery.dataTimestamp` within the `attestationService(...)` function.

Status: Fixed

Update from the client: Fixed in commit 1218e1c

6.9 [Low] Stale yield factor from queued messages can devalue L2 USPD

File(s): /src/BridgeEscrow.sol

Description: The protocol allows USPD tokens to be bridged from L1 to L2 through Wormhole's NTTManager (Native Token Transfer Manager) contract. When tokens are bridged, the message payload includes the token amount and the current L1 yieldFactor. On L2, the releaseShares(...) function in the BridgeEscrow contract mints the bridged tokens and updates the L2's yield factor to match the one sent from L1.

```
function releaseShares(...) external nonReentrant {
// ...
    if (block.chainid == MAINNET_CHAIN_ID) {
        // ...
    } else {
        // L2 (Satellite Chain): Bridging from L1 to L2.
        IcUSPDToken(address(cUSPDToken)).mint(recipient, cUSPDShareAmount);
        totalBridgedInShares += cUSPDShareAmount;

        // @audit The yield factor from the message is used to update the rate contract.
        rateContract.updateL2YieldFactor(l2YieldFactor);
    }
// ...
}
```

The Wormhole NTTManager contract has a rate-limiting feature. If the rate of incoming transfers to an L2 chain goes over a set limit, the transfer message is not handled right away but is placed in a queue. This queued message contains the yieldFactor from when it was started on L1 and can be run later by anyone.

```
// NttManager.sol

function _handleTransfer(
    uint16 sourceChainId,
    bytes32 sourceNttManagerAddress,
    TransceiverStructs.NttManagerMessage memory message,
    bytes32 digest
) internal {
    TransceiverStructs.NativeTokenTransfer memory nativeTokenTransfer =
        TransceiverStructs.parseNativeTokenTransfer(message.payload);

    // verify that the destination chain is valid
    if (nativeTokenTransfer.toChain != chainId) {
        revert InvalidTargetChain(nativeTokenTransfer.toChain, chainId);
    }
    uint8 toDecimals = tokenDecimals();
    TrimmedAmount nativeTransferAmount = (nativeTokenTransfer.amount.untrim(toDecimals)).trim(toDecimals, toDecimals);

    address transferRecipient = fromWormholeFormat(nativeTokenTransfer.to);

    uint256 receivedYield = _handleAdditionalPayload(
        sourceChainId, sourceNttManagerAddress, message.id, message.sender, nativeTokenTransfer
    );

    bool enqueued = _enqueueOrConsumeInboundRateLimit(
        digest, sourceChainId, nativeTransferAmount, transferRecipient, receivedYield
    );

==>       if (enqueued) {
        return;
    }

    _mintOrUnlockToRecipient(
        digest, transferRecipient, nativeTransferAmount, false, sourceChainId, receivedYield
    );
}
```

This creates a situation where a delayed bridge message can update the L2 yieldFactor with a stale value. For example, a transfer is started on L1, but due to rate limits, its execution is queued on L2. While it's in the queue, other transfers are handled, and the L2 yieldFactor is correctly updated. If the queued message with the old (and potentially lower) yieldFactor is executed, it'll overwrite the correct/the most up-to-date yieldFactor value.

This would cause the USPDToken.balanceOf(...) function on L2 to return lower values for all holders, as the balance depends on shares multiplied by the yield factor. If USPD token on L2 is used as collateral in DeFi protocols for lending and borrowing, this sudden change in the USPD balance can lead to liquidation of user positions.

Recommendation(s): Consider implementing staleness checks for `yieldFactor` on L2. This staleness check would prevent an older message from updating the more recent `yieldFactor`.

Status: Acknowledged

Update from the client: We had a staleness check in there that prevented a lower `yieldFactor` being set than the existing one. Now, we removed that, because in an unknown situation where the stETH yield actually gets negative for some reason, we are still stuck with the lockstep for stETH yield and if the stETH yield is negative so is ours. It is going to be a known issue, no fix for the time being.

6.10 [Info] The function setLiquidationParameters should enforce _payoutPercent to be strictly greater than 100

File(s): /src/StabilizerNFT.sol

Description: The `setLiquidationParameters` function allows the admin to update the value of `liquidationLiquidatorPayoutPercent`, which determines the reward payout for liquidators when they liquidate under-collateralized positions:

```
function setLiquidationParameters(uint256 _payoutPercent /* Removed _thresholdPercent */) external
 ↪  onlyRole(DEFAULT_ADMIN_ROLE) {
==>     require(_payoutPercent >= 100, "Payout percent must be >= 100");

    liquidationLiquidatorPayoutPercent = _payoutPercent;
    emit LiquidationParametersUpdated(_payoutPercent);
}
```

Currently, the function permits setting the payout percent to exactly 100%, which would mean liquidators receive no incentive for performing liquidations. This will discourage participation from liquidators.

Recommendation(s): Consider updating the validation check to require `_payoutPercent` to be strictly greater than `100`, ensuring that liquidators always receive a positive incentive for their participation.

Status: Fixed

Update from the client: Fixed in commit 531e688

6.11 [Best Practice] Address is used before being validated

File(s): /src/StabilizerNFT.sol

Description: In the `StabilizerNFT` contract's `liquidatePosition(...)` function, it fetches the current price of `stETH` from an oracle contract. The address of this oracle is retrieved by calling `cuspdToken.oracle()`.

The function first attempts to call the `attestationService(...)` function on the oracle address returned by `cuspdToken.oracle()`. Only after this call does it check whether the returned oracle address is a non-zero address. The addresses should be validated before they are used in interactions.

```
function liquidatePosition(
    // ...
) external {
    // ...
    // @audit The address returned by `cuspdToken.oracle()` is used before it is validated.
    IPriceOracle.PriceResponse memory priceResponse = IPriceOracle(cuspdToken.oracle()).attestationService(priceQuery);
    // @audit The check for a zero address occurs after the address has already been used.
    require(address(cuspdToken.oracle()) != address(0), "Oracle not set in cUSPDToken");
    require(priceResponse.price > 0, "Invalid oracle price");
    // ...
}
```

Recommendation(s): Consider validating the oracle address before using it.

Status: Fixed

Update from the client: Fixed in commit 531e688

6.12 [Best Practice] The shortfall condition in `liquidatePosition` is always true and can safely be removed

File(s): /src/StabilizerNFT.sol

Description: In the `liquidatePosition(...)` function, the logic for distributing collateral to a liquidator is split into two main branches. The `else` block at line is executed only when the actual backing collateral is insufficient to cover the liquidator's target payout (`actualBackingStEth < targetPayoutToLiquidator`).

However, inside this else block, another check is performed:

```
function liquidatePosition(
    // ...
) external {
    // ...
    {
        // ...
        if (actualBackingStEth >= targetPayoutToLiquidator) {
            // ...
        } else {
            // ...

            // @audit This check is always true because the code is in the `else`
            // block of the check above.
==>             if (targetPayoutToLiquidator > actualBackingStEth) {
                uint256 shortfallAmount = targetPayoutToLiquidator - actualBackingStEth;
                // ...
            }
        }
    }
    // ...
}
```

This condition is guaranteed to be true because it is the same condition that caused this branch of the code to be executed in the first place.

This redundant check consumes unnecessary gas and adds a small amount of bytecode.

Recommendation(s): Consider removing the redundant `if` condition.

Status: Fixed

Update from the client: Fixed in commit e20ed01

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 Morpher documentation

`Morpher` team provided an overview of the main system components during the kick-off call with a detailed explanation of the intended functionalities along with a written specification for the different contracts and roles. Moreover, the team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Mutation Testing

The Morpher's test suite is comprehensive and clearly outlines the contract's specification. To assess the test coverage for the audited files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch to "kill" the mutant.

The following table outlines the results of the analysis performed on Morpher's smart contracts. The first column lists the contracts that were tested using mutation testing. The second column indicates the number of mutants generated and how many were "slain" (i.e., caught by the test suite). The third column provides the percentage of mutants slain, reflecting the effectiveness of the test suite in covering the particular contract. The higher the score, the better the test suite is at finding bugs.

Contract	Mutants (slain / total generated)	Score
`src/BridgeEscrow.sol`	62 / 72	86.11%
`src/InsuranceEscrow.sol`	14 / 14	100.00%
`src/OvercollateralizationReporter.sol`	169 / 177	95.48%
`src/PoolSharesConversionRate.sol`	35 / 38	92.10%
`src/PositionEscrow.sol`	182 / 197	92.38%
`src/PriceOracle.sol`	162 / 184	88.04%
`src/StabilizerEscrow.sol`	26 / 31	83.87%
`src/StabilizerNFT.sol`	626 / 800	78.25%
`src/UspdToken.sol`	131 / 141	92.90%
`src/cUSPDToken.sol`	151 / 167	90.41%
Total	1558 / 1821	85.55%

8.2 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 124 files with Solc 0.8.29
[] Solc 0.8.29 finished in 10.88s
Compiler run successful with warnings:
Warning (2072): Unused local variable.
   --> script/04-DeploySystemCore.s.sol:218:13:
    |
218 |             StabilizerNFT stabilizer = StabilizerNFT(payable(stabilizerProxyAddress));
    |             ^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2018): Function state mutability can be restricted to pure
   --> test/PositionEscrow.t.sol:145:5:
    |
145 |     function unallocateStabilizerFunds(
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to pure
  --> test/PriceOracle.t.sol:59:5:
   |
59 |     function _createSignedQueryWithAssetPair(
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to pure
  --> test/mocks/TestPriceOracle.sol:12:5:
   |
12 |     function _isPriceDeviationAcceptable(
   |     ^ (Relevant source part starts here and spans across multiple lines).
```

8.3 Tests Output

```
> forge test
[] Compiling...
[] Compiling 105 files with Solc 0.8.29
[] Solc 0.8.29 finished in 33.85s
Compiler run successful with warnings:
Warning (2018): Function state mutability can be restricted to pure
   --> test/PositionEscrow.t.sol:145:5:
    |
145 |     function unallocateStabilizerFunds(
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to pure
  --> test/PriceOracle.t.sol:59:5:
   |
59 |     function _createSignedQueryWithAssetPair(
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to pure
  --> test/mocks/TestPriceOracle.sol:12:5:
   |
12 |     function _isPriceDeviationAcceptable(
   |     ^ (Relevant source part starts here and spans across multiple lines).


Ran 21 tests for test/BridgeEscrow.t.sol:BridgeEscrowTest
[PASS] test_Integration_LockViaUSPDToken_L1() (gas: 139014)
[PASS] test_Integration_LockViaUSPDToken_L2_BurnsSharesInEscrow() (gas: 150367)
[PASS] test_Integration_UnlockViaUSPDToken_L1() (gas: 158066)
[PASS] test_Integration_UnlockViaUSPDToken_L2_MintsSharesToRecipient() (gas: 112056)
[PASS] test_Revert_DirectEthTransfer() (gas: 14931)
[PASS] test_Unit_EscrowShares_L1_Success() (gas: 69125)
[PASS] test_Unit_EscrowShares_L2_Revert_InsufficientBridgedInShares() (gas: 58510)
[PASS] test_Unit_EscrowShares_L2_Success_BurnsShares() (gas: 121256)
[PASS] test_Unit_EscrowShares_Revert_NotUspdToken() (gas: 15970)
[PASS] test_Unit_EscrowShares_Revert_ZeroAmount() (gas: 18546)
[PASS] test_Unit_RecoverExcessShares_CanBeCalledByAnyone() (gas: 65623)
[PASS] test_Unit_RecoverExcessShares_L1_Success() (gas: 138987)
[PASS] test_Unit_RecoverExcessShares_L2_Success() (gas: 135138)
[PASS] test_Unit_RecoverExcessShares_NoExcess() (gas: 110937)
[PASS] test_Unit_ReleaseShares_L1_Success() (gas: 128986)
[PASS] test_Unit_ReleaseShares_L2_Success_MintsShares() (gas: 99501)
[PASS] test_Unit_ReleaseShares_Revert_InsufficientBridgedShares_PerChain() (gas: 69890)
[PASS] test_Unit_ReleaseShares_Revert_InsufficientBridgedShares_Total() (gas: 89456)
[PASS] test_Unit_ReleaseShares_Revert_NotUspdToken() (gas: 15963)
[PASS] test_Unit_ReleaseShares_Revert_ZeroAmount() (gas: 18651)
[PASS] test_Unit_ReleaseShares_Revert_ZeroRecipient() (gas: 16454)
Suite result: ok. 21 passed; 0 failed; 0 skipped; finished in 40.64ms (7.52ms CPU time)

Ran 11 tests for test/PoolSharesConversionRate.t.sol:PoolSharesConversionRateTest
[PASS] testDeployment() (gas: 15835)
[PASS] testInitialYieldFactor() (gas: 16548)
[PASS] testL2Deployment_And_InitialYieldFactor() (gas: 490262)
[PASS] testRevertIf_Constructor_InitialRateZero() (gas: 1002582)
[PASS] testRevertIf_Constructor_StEthAddressZero() (gas: 60965)
[PASS] testUpdateL2YieldFactor_Revert_AccessControl() (gas: 489034)
[PASS] testUpdateL2YieldFactor_Revert_NotL2Chain() (gas: 40289)
[PASS] testUpdateL2YieldFactor_Success() (gas: 489988)
[PASS] testYieldFactorAfterMultipleRebases() (gas: 60216)
[PASS] testYieldFactorAfterRebase() (gas: 49323)
[PASS] testYieldFactorNoChange() (gas: 27295)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 39.34ms (10.62ms CPU time)
```

```
Ran 18 tests for test/StabilizerEscrow.t.sol:StabilizerEscrowTest
[PASS] test_ApproveAllocation_Revert_InsufficientBalance() (gas: 31596)
[PASS] test_ApproveAllocation_Revert_NotStabilizerNFT() (gas: 20349)
[PASS] test_ApproveAllocation_Revert_ZeroAddress() (gas: 18137)
[PASS] test_ApproveAllocation_Revert_ZeroAmount() (gas: 20230)
[PASS] test_ApproveAllocation_Success() (gas: 63500)
[PASS] test_Deposit_Revert_NotStabilizerNFT() (gas: 25244)
[PASS] test_Deposit_Revert_ZeroAmount() (gas: 17886)
[PASS] test_Deposit_Success() (gas: 68702)
[PASS] test_Initialize_Revert_ZeroLido() (gas: 701827)
[PASS] test_Initialize_Revert_ZeroStETH() (gas: 701779)
[PASS] test_Initialize_Revert_ZeroStabilizerNFT() (gas: 701786)
[PASS] test_Initialize_Success() (gas: 41374)
[PASS] test_Receive_RevertsDirectEthDeposit() (gas: 22915)
[PASS] test_UnallocatedStETH_MatchesBalance() (gas: 199038)
[PASS] test_WithdrawUnallocated_Internal_Revert_InsufficientBalance() (gas: 35628)
[PASS] test_WithdrawUnallocated_Internal_Revert_NotStabilizerNFT() (gas: 17977)
[PASS] test_WithdrawUnallocated_Internal_Revert_ZeroAmount() (gas: 17910)
[PASS] test_WithdrawUnallocated_Internal_Success() (gas: 125439)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 37.72ms (5.46ms CPU time)

Ran 40 tests for test/OvercollateralizationReporter.t.sol:OvercollateralizationReporterTest
[PASS] testInitialization_SnapshotVariables() (gas: 20741)
[PASS] testInitialization_Success() (gas: 43518)
[PASS] testInitialize_Revert_ZeroAdmin() (gas: 1490828)
[PASS] testInitialize_Revert_ZeroCUSPDToken() (gas: 1491009)
[PASS] testInitialize_Revert_ZeroRateContract() (gas: 1490995)
[PASS] testInitialize_Revert_ZeroStabilizerNFT() (gas: 1490943)
[PASS] testUpdateSnapshot_NegativeDelta() (gas: 74599)
[PASS] testUpdateSnapshot_PositiveDelta_ExistingValue() (gas: 74455)
[PASS] testUpdateSnapshot_PositiveDelta_FromZero() (gas: 66874)
[PASS] testUpdateSnapshot_Revert_InconsistentInitialState() (gas: 274100)
[PASS] testUpdateSnapshot_Revert_NotUpdater() (gas: 20081)
[PASS] testUpdateSnapshot_Revert_SnapshotUnderflow() (gas: 62027)
[PASS] testUpdateSnapshot_Success_ForcedZeroInitialYieldAndSnapshot() (gas: 297439)
[PASS] testUpdateSnapshot_WithYieldChange_NegativeDelta() (gas: 112241)
[PASS] testUpdateSnapshot_WithYieldChange_PositiveDelta() (gas: 112073)
[PASS] testUpgrade_Revert_NotUpgrader() (gas: 1420378)
[PASS] testUpgrade_Success() (gas: 1430748)
[PASS] test_GetSystemCollateralizationRatio_Revert_InvalidPriceDecimals() (gas: 121950)
[PASS] test_GetSystemCollateralizationRatio_Revert_ZeroCurrentYieldFactor() (gas: 94529)
[PASS] test_GetSystemCollateralizationRatio_Revert_ZeroOraclePrice() (gas: 121969)
[PASS] test_GetSystemCollateralizationRatio_Success_Typical() (gas: 122551)
[PASS] test_GetSystemCollateralizationRatio_VaryingOutcomes_EqualTo100() (gas: 122536)
[PASS] test_GetSystemCollateralizationRatio_VaryingOutcomes_GreaterThan100() (gas: 122474)
[PASS] test_GetSystemCollateralizationRatio_VaryingOutcomes_LessThan100() (gas: 122498)
[PASS] test_GetSystemCollateralizationRatio_ZeroEstimatedCurrentCollateralStEth() (gas: 94022)
[PASS] test_GetSystemCollateralizationRatio_ZeroLiabilityValueUSD_DueToTruncation() (gas: 243275)
[PASS] test_GetSystemCollateralizationRatio_ZeroTotalShares() (gas: 70203)
[PASS] test_GetSystemCollateralizationRatio_ZeroYieldFactorAtLastSnapshot() (gas: 320319)
[PASS] test_ResetCollateralSnapshot_Revert_NotAdmin() (gas: 19955)
[PASS] test_ResetCollateralSnapshot_Revert_ZeroCurrentYieldFactor() (gas: 42059)
[PASS] test_ResetCollateralSnapshot_Success_Admin() (gas: 76541)
[PASS] test_UpdateCUSPDToken_Revert_NotAdmin() (gas: 2267053)
[PASS] test_UpdateCUSPDToken_Revert_ZeroAddress() (gas: 18517)
[PASS] test_UpdateCUSPDToken_Success_Admin() (gas: 2274770)
[PASS] test_UpdateRateContract_Revert_NotAdmin() (gas: 463823)
[PASS] test_UpdateRateContract_Revert_ZeroAddress() (gas: 18507)
[PASS] test_UpdateRateContract_Success_Admin() (gas: 473424)
[PASS] test_UpdateStabilizerNFTContract_Revert_NotAdmin() (gas: 21593)
[PASS] test_UpdateStabilizerNFTContract_Revert_ZeroAddress() (gas: 18581)
[PASS] test_UpdateStabilizerNFTContract_Success_Admin() (gas: 72966)
Suite result: ok. 40 passed; 0 failed; 0 skipped; finished in 57.30ms (21.57ms CPU time)
```

```
Ran 49 tests for test/PositionEscrow.t.sol:PositionEscrowTest
[PASS] testInitialize() (gas: 63592)
[PASS] test_addCollateralEth_revert_lidoSubmitFails() (gas: 35118)
[PASS] test_addCollateralEth_revert_zeroAmount() (gas: 15824)
[PASS] test_addCollateralEth_success() (gas: 165996)
[PASS] test_addCollateralFromStabilizer_both() (gas: 172191)
[PASS] test_addCollateralFromStabilizer_onlyEth() (gas: 165829)
[PASS] test_addCollateralFromStabilizer_onlyStETH() (gas: 144758)
[PASS] test_addCollateralFromStabilizer_revert_lidoReturnsZero() (gas: 38025)
[PASS] test_addCollateralFromStabilizer_revert_lidoSubmitFails() (gas: 37687)
[PASS] test_addCollateralFromStabilizer_revert_notStabilizerRole() (gas: 27236)
[PASS] test_addCollateralFromStabilizer_revert_zeroInput() (gas: 18373)
[PASS] test_addCollateralStETH_revert_insufficientAllowance() (gas: 165010)
[PASS] test_addCollateralStETH_revert_insufficientBalance() (gas: 165908)
[PASS] test_addCollateralStETH_revert_zeroAmount() (gas: 15905)
[PASS] test_addCollateralStETH_success() (gas: 207851)
[PASS] test_getCollateralizationRatio_normal() (gas: 175060)
[PASS] test_getCollateralizationRatio_withYield() (gas: 177768)
[PASS] test_getCollateralizationRatio_zeroCollateral() (gas: 51576)
[PASS] test_getCollateralizationRatio_zeroShares() (gas: 16275)
[PASS] test_getCurrentStEthBalance() (gas: 135689)
[PASS] test_initialize_revert_zeroLido() (gas: 1626111)
[PASS] test_initialize_revert_zeroOracle() (gas: 1626164)
[PASS] test_initialize_revert_zeroRateContract() (gas: 1626184)
[PASS] test_initialize_revert_zeroStETH() (gas: 1626139)
[PASS] test_initialize_revert_zeroStabilizerNFT() (gas: 1626022)
[PASS] test_initialize_revert_zeroStabilizerOwner() (gas: 1626126)
[PASS] test_modifyAllocation_negativeDelta() (gas: 51243)
[PASS] test_modifyAllocation_positiveDelta() (gas: 45412)
[PASS] test_modifyAllocation_revert_notStabilizerRole() (gas: 19922)
[PASS] test_modifyAllocation_revert_underflow() (gas: 44265)
[PASS] test_modifyAllocation_zeroDelta() (gas: 45680)
[PASS] test_receive_revert_lidoSubmitFails() (gas: 36549)
[PASS] test_receive_success() (gas: 166293)
[PASS] test_receive_zeroAmount() (gas: 27836)
[PASS] test_removeCollateral_revert_insufficientBalance() (gas: 143209)
[PASS] test_removeCollateral_revert_notStabilizerRole() (gas: 22217)
[PASS] test_removeCollateral_revert_transferFails() (gas: 144668)
[PASS] test_removeCollateral_revert_zeroAmount() (gas: 20598)
[PASS] test_removeCollateral_revert_zeroRecipient() (gas: 18473)
[PASS] test_removeCollateral_success() (gas: 221680)
[PASS] test_removeExcessCollateral_revert_belowMinRatioAfterRemoval() (gas: 192967)
[PASS] test_removeExcessCollateral_revert_invalidPriceQuery() (gas: 44875)
[PASS] test_removeExcessCollateral_revert_notManagerRole() (gas: 26428)
[PASS] test_removeExcessCollateral_revert_transferFails() (gas: 194865)
[PASS] test_removeExcessCollateral_revert_zeroOraclePrice() (gas: 192167)
[PASS] test_removeExcessCollateral_revert_zeroRecipient() (gas: 22755)
[PASS] test_removeExcessCollateral_success_excessExists() (gas: 274697)
[PASS] test_removeExcessCollateral_success_noExcess() (gas: 201604)
[PASS] test_removeExcessCollateral_success_zeroLiability() (gas: 202115)
Suite result: ok. 49 passed; 0 failed; 0 skipped; finished in 61.46ms (19.54ms CPU time)
```

```
Ran 25 tests for test/PriceOracle.t.sol:PriceOracleTest
[PASS] testAttestationService_InvalidDecimals() (gas: 63847)
[PASS] testAttestationService_L1_ChainlinkStale() (gas: 78572)
[PASS] testAttestationService_L1_ChainlinkUnavailable() (gas: 78854)
[PASS] testAttestationService_L1_PriceDeviationTooHigh_MorpherVsChainlink() (gas: 93285)
[PASS] testAttestationService_L1_UniswapV3Unavailable() (gas: 82885)
[PASS] testAttestationService_L2Behavior() (gas: 103611)
[PASS] testAttestationService_Revert_InvalidAssetPair() (gas: 56390)
[PASS] testAttestationService_Revert_StaleTimestamp() (gas: 126145)
[PASS] testAttestationService_StalePriceData() (gas: 70034)
[PASS] testAttestationService_WhenPaused() (gas: 79116)
[PASS] testInitialSetup() (gas: 29622)
[PASS] testInitialize_Revert_MaxDeviationTooHigh() (gas: 1955155)
[PASS] testInitialize_Revert_StalenessPeriodTooHigh() (gas: 1955115)
[PASS] testInitialize_Revert_ZeroAdmin() (gas: 1941748)
[PASS] testInitialize_Revert_ZeroChainlinkAggregator() (gas: 1943870)
[PASS] testInitialize_Revert_ZeroUniswapRouter() (gas: 1943793)
[PASS] testInitialize_Revert_ZeroUsdcAddress() (gas: 1943743)
[PASS] testSetMaxDeviationPercentage_Revert_NotAdmin() (gas: 19987)
[PASS] testSetMaxDeviationPercentage_Revert_TooHigh() (gas: 18774)
[PASS] testSetMaxDeviationPercentage_Success() (gas: 24545)
[PASS] testSetPriceStalenessPeriod_Revert_NotAdmin() (gas: 20020)
[PASS] testSetPriceStalenessPeriod_Revert_TooHigh() (gas: 18643)
[PASS] testSetPriceStalenessPeriod_Success() (gas: 24568)
[PASS] testUnauthorizedSigner() (gas: 27434)
[PASS] testUnpause() (gas: 66502)
Suite result: ok. 25 passed; 0 failed; 0 skipped; finished in 65.00ms (30.81ms CPU time)

Ran 31 tests for test/cUSPDToken.t.sol:cUSPDTokenTest
[PASS] testBurnShares_Revert_InsufficientBalance() (gas: 1271152)
[PASS] testBurnShares_Revert_InvalidPrice() (gas: 1261585)
[PASS] testBurnShares_Revert_NoAllocatedStabilizers() (gas: 156393)
[PASS] testBurnShares_Revert_TransferFailed() (gas: 1410680)
[PASS] testBurnShares_Revert_ZeroAmount() (gas: 17653)
[PASS] testBurnShares_Revert_ZeroRecipient() (gas: 15496)
[PASS] testBurnShares_Success() (gas: 1446280)
[PASS] testBurn_Success_OnL2() (gas: 126023)
[PASS] testBurn_Success_OnMainnet() (gas: 96842)
[PASS] testBurn_Success_OnSepolia() (gas: 96919)
[PASS] testConstructor_Revert_ZeroAddresses() (gas: 343198)
[PASS] testConstructor_Success() (gas: 47717)
[PASS] testMintShares_Revert_InvalidPrice() (gas: 29845)
[PASS] testMintShares_Revert_InvalidYield() (gas: 99492)
[PASS] testMintShares_Revert_NoEthSent() (gas: 15500)
[PASS] testMintShares_Revert_NoUnallocatedStabilizers() (gas: 131717)
[PASS] testMintShares_Revert_ZeroRecipient() (gas: 22699)
[PASS] testMintShares_Success() (gas: 1263132)
[PASS] testMint_Revert_OnMainnet() (gas: 44180)
[PASS] testMint_Revert_OnSepolia() (gas: 44091)
[PASS] testMint_Success_OnL2() (gas: 93660)
[PASS] testReceive_Reverts() (gas: 14818)
[PASS] testUpdateOracle_Revert_NotUpdater() (gas: 16120)
[PASS] testUpdateOracle_Revert_ZeroAddress() (gas: 13507)
[PASS] testUpdateOracle_Success() (gas: 26012)
[PASS] testUpdateRateContract_Revert_NotUpdater() (gas: 16119)
[PASS] testUpdateRateContract_Revert_ZeroAddress() (gas: 13533)
[PASS] testUpdateRateContract_Success() (gas: 26160)
[PASS] testUpdateStabilizer_Revert_NotUpdater() (gas: 16131)
[PASS] testUpdateStabilizer_Revert_ZeroAddress() (gas: 13571)
[PASS] testUpdateStabilizer_Success() (gas: 26055)
Suite result: ok. 31 passed; 0 failed; 0 skipped; finished in 62.41ms (43.25ms CPU time)

Ran 10 tests for test/InsuranceEscrow.t.sol:InsuranceEscrowTest
[PASS] test_GetStEthBalance() (gas: 111140)
[PASS] test_RevertIf_Constructor_ZeroStETHAddress() (gas: 62225)
[PASS] test_RevertIf_DepositStEth_TransferFromReturnsFalse() (gas: 1032213)
[PASS] test_RevertIf_DepositStEth_ZeroAmount() (gas: 10563)
[PASS] test_RevertIf_ReceiveEth() (gas: 15295)
[PASS] test_RevertIf_WithdrawStEth_TransferReturnsFalse() (gas: 1008561)
[PASS] test_RevertIf_WithdrawStEth_ZeroAmount() (gas: 12969)
[PASS] test_RevertIf_WithdrawStEth_ZeroToAddress() (gas: 10791)
[PASS] test_Successful_DepositStEth() (gas: 112344)
[PASS] test_Successful_WithdrawStEth() (gas: 193659)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 72.59ms (25.64ms CPU time)
```

```
Ran 42 tests for test/UspdToken.t.sol:USPDTokenTest
[PASS] testAdminRoleAssignment() (gas: 11836)
[PASS] testAllowance_SameIfYieldFactorIsZero() (gas: 54683)
[PASS] testApproveAndAllowance_Success() (gas: 38625)
[PASS] testApprove_Success_ZeroYieldFactor() (gas: 53055)
[PASS] testBalanceOf_ZeroIfCuspdTokenIsZero() (gas: 142624)
[PASS] testBalanceOf_ZeroIfRateContractIsZero() (gas: 140604)
[PASS] testBalanceOf_ZeroIfYieldFactorIsZero() (gas: 37513)
[PASS] testConstructor_Revert_ZeroAdminAddress() (gas: 90176)
[PASS] testConstructor_Revert_ZeroCUSPDAddress() (gas: 88010)
[PASS] testConstructor_Revert_ZeroRateContractAddress() (gas: 88083)
[PASS] testLockForBridging_Revert_AccessControl() (gas: 42510)
[PASS] testLockForBridging_Revert_AmountTooSmallForHighYield() (gas: 1457729)
[PASS] testLockForBridging_Revert_BridgeEscrowNotSet() (gas: 43542)
[PASS] testLockForBridging_Revert_ZeroYieldFactor() (gas: 87768)
[PASS] testMintByDirectEtherTransfer() (gas: 22769)
[PASS] testMint_Revert_CUSPDTokenAddressZero() (gas: 151530)
[PASS] testMint_Revert_InvalidPriceQuery() (gas: 81027)
[PASS] testMint_Revert_MintToZeroAddress() (gas: 48430)
[PASS] testMint_Revert_NoUnallocatedFunds() (gas: 147803)
[PASS] testMint_Revert_ZeroEthSent() (gas: 48836)
[PASS] testMint_Success_ExactEthNoRefund() (gas: 1270664)
[PASS] testMint_Success_FullAllocation() (gas: 1286221)
[PASS] testSetBridgeEscrowAddress() (gas: 51140)
[PASS] testTotalSupply_Success() (gas: 1284680)
[PASS] testTotalSupply_ZeroIfCuspdTokenIsZero() (gas: 140922)
[PASS] testTotalSupply_ZeroIfRateContractIsZero() (gas: 138924)
[PASS] testTotalSupply_ZeroIfYieldFactorIsZero() (gas: 36200)
[PASS] testTransferFrom_Revert_AmountTooSmallForHighYield() (gas: 1430182)
[PASS] testTransferFrom_Revert_ZeroYieldFactor() (gas: 39083)
[PASS] testTransferFrom_Success() (gas: 1339560)
[PASS] testTransferFrom_Success_ZeroAmount() (gas: 1322699)
[PASS] testTransfer_Revert_AmountTooSmallForHighYield() (gas: 1402711)
[PASS] testTransfer_Revert_ZeroAmount() (gas: 1286725)
[PASS] testTransfer_Revert_ZeroYieldFactor() (gas: 37777)
[PASS] testTransfer_Success() (gas: 1324589)
[PASS] testUnlockFromBridging_Revert_AccessControl() (gas: 44179)
[PASS] testUnlockFromBridging_Revert_AmountTooSmall() (gas: 71088)
[PASS] testUnlockFromBridging_Revert_BridgeEscrowNotSet() (gas: 45261)
[PASS] testUnlockFromBridging_Revert_ZeroRecipient() (gas: 68497)
[PASS] testUnlockFromBridging_Revert_ZeroSourceYieldFactor() (gas: 69949)
[PASS] testUpdateCUSPDAddress() (gas: 35048)
[PASS] testUpdateRateContract() (gas: 35038)
Suite result: ok. 42 passed; 0 failed; 0 skipped; finished in 72.64ms (78.90ms CPU time)
```

```
Ran 69 tests for test/StabilizerNFT.t.sol:StabilizerNFTTest
[PASS] testAddUnallocatedFundsEth_Multiple() (gas: 911578)
[PASS] testAddUnallocatedFundsEth_Revert_NonExistentToken() (gas: 25791)
[PASS] testAddUnallocatedFundsEth_Revert_NotOwner() (gas: 686272)
[PASS] testAddUnallocatedFundsEth_Revert_ZeroAmount() (gas: 676925)
[PASS] testAddUnallocatedFundsEth_Success() (gas: 880760)
[PASS] testAddUnallocatedFundsStETH_Revert_InsufficientAllowance() (gas: 824459)
[PASS] testAddUnallocatedFundsStETH_Revert_InsufficientBalance() (gas: 825121)
[PASS] testAddUnallocatedFundsStETH_Revert_NonExistentToken() (gas: 19117)
[PASS] testAddUnallocatedFundsStETH_Revert_NotOwner() (gas: 824371)
[PASS] testAddUnallocatedFundsStETH_Revert_ZeroAmount() (gas: 676927)
[PASS] testAddUnallocatedFundsStETH_Success() (gas: 871075)
[PASS] testAllocateStabilizerFunds_GasExhaustionInLoop() (gas: 39017500)
[PASS] testAllocateStabilizerFunds_LoopSkip_RemainingEthZero() (gas: 1822261)
[PASS] testAllocateStabilizerFunds_PartialAllocation_UserEthLimited() (gas: 2919181)
[PASS] testAllocateStabilizerFunds_ReporterInteraction_SuccessfulAllocation() (gas: 1259579)
[PASS] testAllocateStabilizerFunds_Revert_NoEthSent() (gas: 879034)
[PASS] testAllocateStabilizerFunds_Revert_NoFundsCanBeAllocated() (gas: 783646)
[PASS] testAllocateStabilizerFunds_Revert_NotCUSPDToken() (gas: 25784)
[PASS] testAllocateStabilizerFunds_SomeEscrowsEmpty() (gas: 2774770)
[PASS] testAllocateStabilizerFunds_UserEthExceedsAllStabilizerCapacity() (gas: 2047111)
[PASS] testAllocatedAndUnallocatedIds() (gas: 3019984)
[PASS] testAllocationAndPositionNFT() (gas: 1256747)
[PASS] testInitialize_Revert_ZeroInsuranceEscrow() (gas: 7463599)
[PASS] testInitialize_Revert_ZeroPositionEscrowImpl() (gas: 6455295)
[PASS] testInitialize_Revert_ZeroStabilizerEscrowImpl() (gas: 7371935)
[PASS] testLiquidation_NoReporterSet() (gas: 1368628)
[PASS] testLiquidation_PartialLiquidation_PositionRemainsAllocated() (gas: 1353470)
[PASS] testLiquidation_PrivilegedVsDefaultThreshold() (gas: 3062907)
[PASS] testLiquidation_Success_BelowThreshold_FullPayoutFromCollateral() (gas: 2210814)
[PASS] testLiquidation_Success_BelowThreshold_RemainderToInsurance() (gas: 2300212)
[PASS] testLiquidation_Success_InsufficientCollateral_InsuranceCoversFullShortfall() (gas: 1404897)
[PASS] testLiquidation_Success_InsufficientCollateral_InsuranceCoversPartialShortfall() (gas: 1351642)
[PASS] testLiquidation_Success_NoCollateralInPosition_InsuranceCoversFullPayout() (gas: 1454349)
[PASS] testLiquidation_Success_NoCollateralInPosition_InsuranceCoversPartialPayout() (gas: 1456632)
[PASS] testLiquidation_Success_NoCollateral_NoInsurance_NoPayout() (gas: 1360336)
[PASS] testLiquidation_WithLiquidatorNFT_HighID_UsesMinThreshold() (gas: 20359165)
[PASS] testLiquidation_WithLiquidatorNFT_ID1_Uses125PercentThreshold() (gas: 1983177)
[PASS] testListManagement_MiddleInsertion() (gas: 3164103)
[PASS] testListManagement_MiddleRemoval() (gas: 3228332)
[PASS] testMintDeploysEscrow() (gas: 736275)
[PASS] testMint_Revert_NonL1ChainId() (gas: 17963)
[PASS] testMultipleStabilizersAllocation() (gas: 2135875)
[PASS] testRemoveUnallocatedFunds_Revert_InsufficientBalance() (gas: 876935)
[PASS] testRemoveUnallocatedFunds_Revert_NonExistentToken() (gas: 18616)
[PASS] testRemoveUnallocatedFunds_Revert_NotOwner() (gas: 874429)
[PASS] testRemoveUnallocatedFunds_Revert_ZeroAmount() (gas: 677059)
[PASS] testRemoveUnallocatedFunds_Success() (gas: 967093)
[PASS] testRemoveUnallocatedFunds_Success_EmptyEscrow() (gas: 864613)
[PASS] testReportCollateralAddition_Success() (gas: 734881)
[PASS] testReportCollateralRemoval_Success() (gas: 742797)
[PASS] testReportCollateral_Revert_NoRole() (gas: 25458)
[PASS] testReportCollateral_Revert_ZeroYieldFactor() (gas: 723402)
[PASS] testReportCollateral_ZeroAmount_NoAction() (gas: 696081)
[PASS] testSetBaseURI_Revert_NotAdmin() (gas: 20727)
[PASS] testSetBaseURI_Success() (gas: 695189)
[PASS] testSetInsuranceEscrow_Revert_NotAdmin() (gas: 21624)
[PASS] testSetInsuranceEscrow_Revert_ZeroAddress() (gas: 18401)
[PASS] testSetInsuranceEscrow_Success() (gas: 29633)
[PASS] testSetLiquidationParameters_Revert_NotAdmin() (gas: 20116)
[PASS] testSetLiquidationParameters_Revert_PayoutTooLow() (gas: 18399)
[PASS] testSetLiquidationParameters_Success() (gas: 27589)
[PASS] testSetMinCollateralizationRatio() (gas: 687893)
[PASS] testTokenURI_EmptyBaseURI() (gas: 688494)
[PASS] testUnallocateStabilizerFunds_Revert_NoFundsUnallocated_AmountTooSmall() (gas: 1182153)
[PASS] testUnallocateStabilizerFunds_Undercollateralized_NoInsurance() (gas: 2262985)
[PASS] testUnallocateStabilizerFunds_Undercollateralized_WithInsuranceCoverage() (gas: 2344863)
[PASS] testUnallocationAndPositionNFT() (gas: 1446115)
[PASS] testUpgradeStabilizerNFT_Revert_NotUpgrader() (gas: 5003378)
[PASS] testUpgradeStabilizerNFT_Success() (gas: 5010179)
Suite result: ok. 69 passed; 0 failed; 0 skipped; finished in 78.25ms (197.02ms CPU time)

Ran 10 test suites in 628.41ms (587.34ms CPU time): 316 tests passed, 0 failed, 0 skipped (316 total tests)
```

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.