Security Review Report
NM-0094 Gyroscope

NETHERMIND

(Aug 15, 2023)

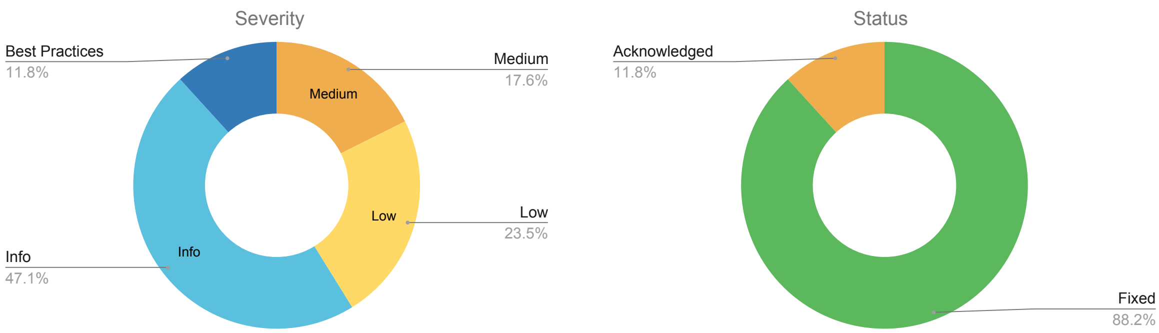
Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind for the Gyroscope protocol. `Gyroscope` is a new stablecoin design that, like a physical gyroscope, remains stable as the surrounding environment changes. The `Gyroscope` stablecoin aims at a long-term reserve ratio of 100%, where every unit of stablecoin is backed by 1 USD worth of collateral. The reserve is a basket of protocol-controlled assets that jointly collateralize the issued stablecoin. The reserve aims to diversify all risks in DeFi to the greatest extent possible. It considers not only price risk but also censorship, regulatory, counterparty, oracle, and governance risks. Prices for minting and redeeming stablecoins are set autonomously to balance the goal of maintaining a tight peg with the goal of the long-term viability of the project in the face of short-term crises.

The audited code comprises 4670 lines of code in Solidity. The `Gyroscope` team has provided detailed documentation explaining the protocol summary, the math behind their Primary Market Market, and the multiple existing mechanisms for protecting and balancing the protocol reserves. **The audit was performed using**: (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts. **Along this document, we report** 17 points of attention, where three are classified as `Medium`, three are classified as `Low`, and eleven are classified as `Informational` or `Best Practice`. 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 (0), **High** (0), **Medium** (3), **Low** (3), **Undetermined** (0), **Informational** (8), **Best Practices** (3).
b) Distribution of status: Fixed (15), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	July 24, 2023
Final Report	August 15, 2023
Methods	Manual Review, Automated Analysis
Repository	protocol
Commit Hash	39066d89d3ff9476d65dfb2ca9059c247379d7db
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/GydToken.sol	47	2	4.3%	12	61
2	contracts/GydRecovery.sol	292	48	16.4%	62	402
3	contracts/ReserveManager.sol	137	11	8.0%	23	171
4	contracts/Motherboard.sol	521	56	10.7%	95	672
5	contracts/GyroConfig.sol	120	15	12.5%	22	157
6	contracts/FreezableProxy.sol	16	6	37.5%	5	27
7	contracts/LiquidityMining.sol	102	12	11.8%	20	134
8	contracts/PrimaryAMMV1.sol	479	45	9.4%	81	605
9	contracts/Reserve.sol	35	8	22.9%	10	53
10	contracts/ReserveStewardshipIncentives.sol	171	21	12.3%	40	232
11	contracts/VaultRegistry.sol	88	7	8.0%	21	116
12	contracts/oracles/CheckedPriceOracle.sol	325	19	5.8%	70	414
13	contracts/oracles/BatchVaultPriceOracle.sol	80	6	7.5%	15	101
14	contracts/oracles/GenericVaultPriceOracle.sol	10	3	30.0%	2	15
15	contracts/oracles/BaseVaultPriceOracle.sol	18	5	27.8%	5	28
16	contracts/oracles/AssetRegistry.sol	101	16	15.8%	26	143
17	contracts/oracles/TellorOracle.sol	29	2	6.9%	5	36
18	contracts/oracles/ChainLinkPriceOracle.sol	28	6	21.4%	7	41
19	contracts/oracles/TrustedSignerPriceOracle.sol	95	17	17.9%	22	134
20	contracts/oracles/BaseChainLinkOracle.sol	33	2	6.1%	9	44
21	contracts/oracles/balancer/BaseBalancerPriceOracle.sol	19	4	21.1%	5	28
22	contracts/oracles/balancer/BalancerLPSharePricing.sol	225	124	55.1%	38	387
23	contracts/oracles/balancer/BalancerCPMMPriceOracle.sol	21	3	14.3%	5	29
24	contracts/oracles/balancer/BalancerECLPV2PriceOracle.sol	29	3	10.3%	7	39
25	contracts/oracles/balancer/Balancer2CLPPriceOracle.sol	23	3	13.0%	5	31
26	contracts/oracles/balancer/Balancer3CLPPriceOracle.sol	21	3	14.3%	5	29
27	contracts/auth/GovernableBase.sol	23	4	17.4%	5	32
28	contracts/auth/Governable.sol	8	2	25.0%	2	12
29	contracts/auth/GovernableUpgradeable.sol	12	3	25.0%	3	18
30	contracts/auth/GovernanceProxy.sol	9	2	22.2%	3	14
31	contracts/safety/ReserveSafetyManager.sol	243	60	24.7%	56	359
32	contracts/safety/VaultSafetyMode.sol	198	14	7.1%	37	249
33	contracts/safety/RootSafetyCheck.sol	76	9	11.8%	17	102
34	contracts/fee_handlers/StaticPercentageFeeHandler.sol	50	8	16.0%	13	71
35	contracts/vaults/BalancerPoolVault.sol	27	7	25.9%	8	42
36	contracts/vaults/GenericVault.sol	18	4	22.2%	4	26
37	contracts/vaults/BaseVault.sol	106	17	16.0%	25	148
38	contracts/read_only/ReserveSystemRead.sol	44	3	6.8%	10	57
39	libraries/Vaults.sol	11	6	54.5%	1	18
40	libraries/ConfigHelpers.sol	109	3	2.8%	23	135
41	libraries/Errors.sol	55	7	12.7%	7	69
42	libraries/VaultMetadataExtension.sol	27	2	7.4%	5	34
43	libraries/Arrays.sol	44	6	13.6%	4	54
44	libraries/DecimalScale.sol	22	2	9.1%	3	27
45	libraries/ConfigKeys.sol	31	5	16.1%	7	43
46	libraries/DataTypes.sol	114	24	21.1%	19	157
47	libraries/TypeConversion.sol	35	5	14.3%	4	44
48	libraries/ReserveStateExtensions.sol	61	6	9.8%	8	75
49	libraries/Flow.sol	20	3	15.0%	3	26
50	libraries/SignedFixedPoint.sol	49	52	106.1%	21	122
	Total	**4457**	**701**	**15.7%**	**905**	**6063**

Changes applied in this Pull Request are also in scope for this assessment.

3 Summary of Issues

	Finding	Severity	Update
1	First depositor can break the minting of shares	Medium	Fixed
2	Order of `vaultInfo.pricedTokens` is not checked	Medium	Fixed
3	Users might not receive rewards for the last period before `rewardsEmissionEndTime`	Medium	Fixed
4	Users can bypass the `perUserSupplyCap` limit during minting	Low	Fixed
5	Users can bypass the `externalCallWhitelist` and execute arbitrary logic by using the "permit" functionality	Low	Fixed
6	`ExternalActionExecutor` could implement an address whitelist	Info	Acknowledged
7	`dryMint(...)` result differs from `mint(...)` result	Low	Fixed
8	Dead code in `VaultRegistry`	Info	Fixed
9	Excessive memory allocation in the function `batchRelativePriceCheck(...)`	Info	Fixed
10	Region detection does not exclude equality cases	Info	Fixed
11	Return value of 0 from `ecrecover` is not checked	Info	Fixed
12	Users can execute calls from the Motherboard contract	Info	Fixed
13	Users might lose everything if they do not call the function `withdraw(...)` when a pending withdrawal is available immediately	Info	Acknowledged
14	XL value in the code and paper is different for `isInSecondRegion(...)`	Info	Fixed
15	Interface `AggregatorV2V3Interface` has multiple functions from deprecated Chainlink API	Best Practices	Fixed
16	Returning the named returns is redundant	Best Practices	Fixed
17	Upgradeability issues	Best Practices	Fixed

4 System Overview

The main contracts of the system included:

a) **Motherboard**

b) **ReserveManager**

c) **GyroConfig**

d) **GydToken**

e) **PrimaryAMMV1**

f) **RootPriceOracle**

g) **RootSafetyCheck**

h) **Reserve**

i) **VaultRegistry**

j) **GydRecovery**

k) **ReserveStewardshipIncentives**

Fig. 2 presents the interaction diagram of the contracts.

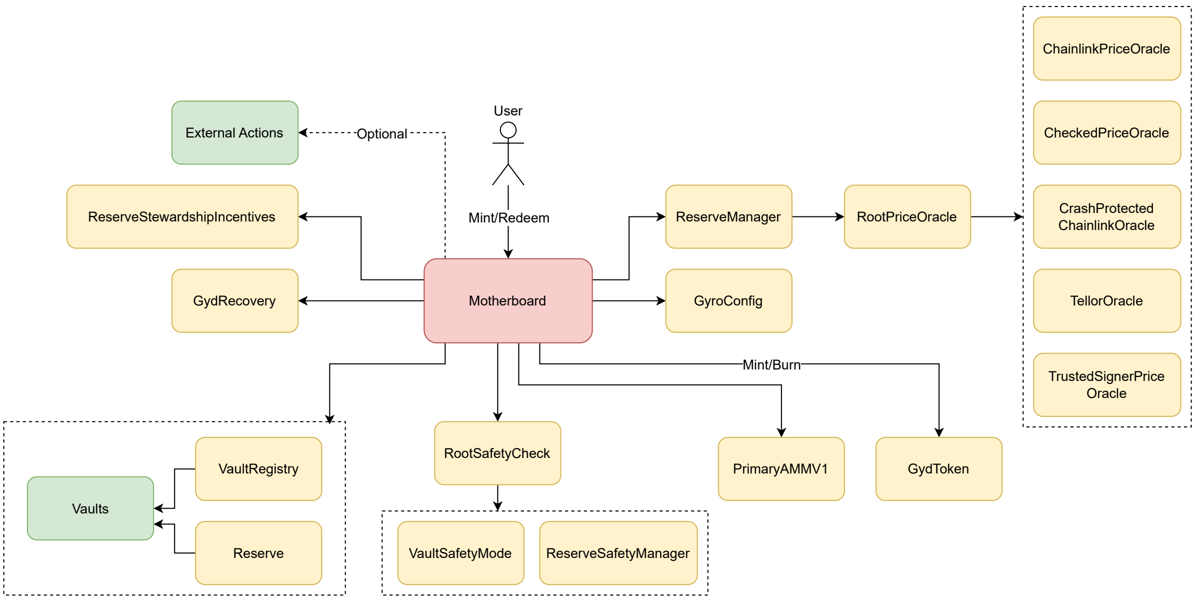


Fig. 2: Interaction Diagram of Contracts

The **Gyroscope** protocol is designed to integrate multiple contracts, each of which seamlessly plugs into the central **Motherboard** contract. The **Motherboard** contract also serves as the primary interface through which users can interact to perform essential actions like minting or redeeming the GYD stablecoin.

The **ReserveManager** acts as the platform where the governance can establish the list of approved Vaults to be used as collateral for minting the GYD token. This contract also includes an important function called `getReserveState()` that plays a crucial role during the minting and redeeming processes. It provides essential information about the system's current status, including the price and weights of all the vaults involved. This information is used internally to ensure the smooth functioning of the minting and redeeming operations, helping to maintain the stability and reliability of the GYD token ecosystem.

The **GyroConfig** contract is the central repository holding all the configurations and their associated metadata for the Gyroscope protocol. It plays a crucial role in storing essential information, including the addresses of various components within the protocol, such as the ReserveManager, Recovery Modules, Safety Check, and other critical elements.

The **GydToken** is fundamentally an ERC20 token intentionally designed to maintain a pegged value of 1 dollar. Moreover, the protocol's governance has the authority to manage the list of minters by adding and removing them.

The **PrimaryAMMV1** contract implements the primary AMM pricing mechanism within the protocol. It is crucial in determining the pricing dynamics for minting and redeeming actions. When the redemption outflow becomes excessive, the bonding curve in the PrimaryAMMV1 contract acts as a circuit breaker mechanism. It dynamically decreases redemption quotes, creating a disincentive for potential runs and attacks on the currency peg.

The **RootPriceOracle** contract acts as the central hub through which various components within the ecosystem interact to obtain price data. It serves as the primary interface for fetching price information from external sources. The **RootPriceOracle** contract then calls different price oracles, each responsible for providing data on specific assets or tokens.

The **RootSafetyCheck** contract serves as the primary interface that other components within the system utilize to perform various safety checks. These safety checks are crucial to ensure the overall integrity and stability of the ecosystem. Components can call the **RootSafetyCheck** contract to execute safety checks on critical aspects of the system, such as the Reserve and Vault safety checks. These checks help assess the health and risk factors associated with the reserve funds and the individual vaults used for collateralization.

The **Reserve** contract primarily functions as a secure repository for holding the collateral users provide when they mint GYD tokens. Users who contribute liquidity to the Vault receive vault tokens in return. These newly minted vault tokens are transferred and kept in the Reserve contract.

The **VaultRegistry** keeps essential information about each Vault within the protocol. This includes crucial details such as the flow, price, and weight associated with each specific Vault. To ensure proper governance and oversight, the functions within the **VaultRegistry** contract can only be accessed and executed by either the Reserve Manager or the governance.

The **GydRecovery** contract allows users to stake their GYD tokens and receive rewards in return. By staking their GYD tokens, users contribute to a pool of funds that will be utilized if the GYD token becomes depegged from its intended value.

The **ReserveStewardshipIncentives** contract serves as a mechanism to incentivize and reward good governance practices within the protocol. When the governance ensures that the reserve health is well-maintained and does not violate predefined thresholds during a specified period, they become eligible to receive rewards in the form of GYD tokens.

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 [Medium] First depositor can break the minting of shares

File(s): BaseVault.sol

Description: In the context of vaults, there is a recurring issue that allows the first depositor to exploit the share minting process, gaining an unfair advantage. The problem occurs when the initial depositor submits a single wei of the underlying token to the Vault using the `deposit()` function. Subsequently, they can contribute a significantly larger amount of the underlying token directly to the vault, artificially inflating the share ratio. For example, they can transfer `1e26 wei` directly to the Vault. Since the `totalSupply()` remains at `1`, they can still withdraw all the funds. However, when subsequent depositors contribute an amount smaller than `1e26 wei`, the calculation for share allocation may round down to zero, effectively minting no shares for them.

Recommendation(s): Consider implementing the following two recommendations:

1. During the initial mint, the contract mints the first 1000 shares to the zero address;
2. The contract should enforce that the minted shares are not zero;

Status: Fixed.

Update from the client: Fixed in commit 7984e27b5e504157560e6f27509732e2417df32e.

6.2 [Medium] Order of `vaultInfo.pricedTokens` is not checked

File(s): BatchVaultPriceOracle.sol

Description: The `_assignUnderlyingTokenPrices(...)` function fills in token prices from the `VaultInfo` structure for `ReserveState` construction. This function assumes that the `pricedTokens` array is sorted. The assumption enables the optimization within the function, as demonstrated below:

```
    function _assignUnderlyingTokenPrices(
        DataTypes.VaultInfo memory vaultInfo,
        address[] memory tokens,
        uint256[] memory underlyingPrices
    ) internal pure {
        for ((uint256 i, uint256 j) = (0, 0); i < vaultInfo.pricedTokens.length; i++) {
            // @audit - Order of `vaultInfo.pricedTokens` is never checked.
            // Here we make use of the fact that both vaultInfo.pricedTokens and tokens are sorted by
            // token address, so we don't have to reset j.
            while (tokens[j] != vaultInfo.pricedTokens[i].tokenAddress) j++;
            vaultInfo.pricedTokens[i].price = underlyingPrices[j];
        }
    }
```

Nonetheless, no mechanism exists to verify the order of `vaultInfo.pricedTokens`. Should this array be unordered, the `_assignUnderlyingTokenPrices(...)` function could potentially fail, blocking retrieval of the current `ReserveState`. This, in turn, would hinder primary actions such as `mint` and `redeem`. This issue is not triggered due to Gyroscope's use of external vaults, which enforces this property internally. However, the assumption may not hold if different types of vaults are supported in the future.

Recommendation(s): It is advisable to introduce protocol-level checks to ensure the `pricedTokens` array remains consistently sorted. This will help prevent potential future issues related to an unordered array.

Status: Fixed.

Update from the client: Fixed in commit e890f541a49fc6e92e23d3b3e3605afce3cb9b09.

6.3 [Medium] Users might not receive rewards for the last period before `rewardsEmissionEndTime`

File(s): `LiquidityMining.sol`

Description: The `LiquidityMining` contract distributes a fixed amount of reward tokens every second from the start of mining until the `rewardsEmissionEndTime` is reached. The `rewardsEmissionRate(...)` function returns the current reward rate, which is based on the time when it is called. If the current time is after the `rewardsEmissionEndTime`, the reward rate is set to `0`. However, the rewards are only accumulated when someone interacts with the contract. This means that when `rewardsEmissionRate(...)` is called after the mining period has ended, the reward rate will be `0` for the last period before it ended as well. As a result, users who have staked in the contract may lose some rewards if they are unaware of this behavior. For example, if the last checkpoint time is `t1` and `rewardsEmissionEndTime = t2`, the amount of reward during `[t1, t2]` will not be distributed to users.

```
function rewardsEmissionRate() public view override returns (uint256) { // @audit Reward rate suddenly changes
    return block.timestamp <= rewardsEmissionEndTime ? _rewardsEmissionRate : 0;
}
```

Recommendation(s): Consider calculating the accumulated amount directly instead of returning only the emission rate. This will provide the function `globalCheckpoint(...)` with more accurate information about the rewards emitted since the last checkpoint time.

Status: Fixed.

Update from the client: Fixed in e63a67761fdbaa13afbe87fda5abf5bafa5b3b6e.

6.4 [Low] Users can bypass the `perUserSupplyCap` limit during minting

File(s): `Motherboard.sol`

Description: In the `Motherboard` contract, there is a `perUserSupplyCap` that limits the amount of GYD an account can mint. However, the current implementation allows users to potentially bypass this limit by transferring their GYD balance to another wallet and then continuing the minting process. The relevant code snippet is provided below:

```
function _isOverCap(address account, uint256 mintedGYDAmount) internal view returns (bool) {
    uint256 globalSupplyCap = gyroConfig.getGlobalSupplyCap();
    if (gydToken.totalSupply() + mintedGYDAmount > globalSupplyCap) {
        return true;
    }
    bool isAuthenticated = gyroConfig.isAuthenticated(account);
    uint256 perUserSupplyCap = gyroConfig.getPerUserSupplyCap(isAuthenticated);
    return gydToken.balanceOf(account) + mintedGYDAmount > perUserSupplyCap; // @audit user can easily bypass this cap
}
```

Recommendation(s): Consider reviewing the logic for checking the `perUserSupplyCap` or removing this limit altogether, as the current implementation allows users to bypass it by transferring their GYD balance to another wallet.

Status: Fixed.

Update from the client: Fixed in commit 4765a6db1d3aabf8f4a378f0d5703ae49ae54ef3.

6.5 [Low] Users can bypass the `externalCallWhitelist` and execute arbitrary logic by using the `permit` functionality

File(s): `Motherboard.sol`

Description: Within the `Motherboard` contract, there is a scenario where users can perform external calls before the minting process. However, these calls are restricted to a whitelist, the `externalCallWhitelist`. Despite this restriction, users can circumvent the check using the `permit` feature. The issue lies in the `_executePermits()` function, which fails to validate whether the `permit.target` is a valid ERC20 token or not. Consequently, users can deploy a contract and inject their custom logic into the `permit()` function, enabling them to execute any desired logic before the minting operation.

```
function _executePermits(DataTypes.PermitData[] calldata permits) internal {
    for (uint256 i = 0; i < permits.length; i++) {
        DataTypes.PermitData calldata permit = permits[i];
        IERC20Permit(permit.target).safePermit(
            permit.owner,
            permit.spender,
            permit.amount,
            permit.deadline,
            permit.v,
            permit.r,
            permit.s
        );
    }
}
```

Recommendation(s): Consider adding a check for `permit.target` before executing the call. One possible approach is introducing a whitelisted set, similar to the existing `externalCallWhitelist`, to ensure that only permitted targets can be invoked.

Status: Fixed.

Update from the client: Fixed in 2f5df68cffd7cacb555b5efd07e85b1c8e7effc9.

6.6 [Low] `ExternalActionExecutor` could implement an address whitelist

File(s): `Motherboard.sol`

Description: The `Motherboard`'s version reviewed in this audit permitted users to execute external calls before the minting process. These calls were restricted to a whitelist, the `externalCallWhitelist`. As a recommendation, we suggested using another contract with no privileged role to execute these external calls. The commit 2f5df68cffd7cacb555b5efd07e85b1c8e7effc9 contains the fixes recommended during the audit with the removal of the `externalCallWhitelist`. We consider the introduced `ExternalActionExecutor` contract could add a whitelisted set of addresses that can be called, which might reduce potential risks to the protocol. Generally, we consider a better design to have a model that does not allow users to conduct arbitrary calls through the protocol. If specific use cases are defined for the feature, it may be better to limit calls to those use cases.

Status: Acknowledged.

Update from the client: We discussed internally and decided to accept the risk. Since this contract has no permission, there is no security risk involved. Protocols allowing users to perform arbitrary actions are very common (e.g., flash loans), and we are unaware of any legal repercussions or damage to the given protocols. Therefore, we would rather keep things as-is.

6.7 [Low] `dryMint(...)` result differs from `mint(...)` result

File(s): `Motherboard.sol`

Description: The `dryMint(...)` function simulates a mint to know whether it would succeed and how much would be minted. However, it may return a different result under certain conditions than a call to `mint(...)`. Both of these functions will execute certain safety checks before minting, as seen in the following code snippet.

```
function mint(...) public override returns (...)
{
    ///////////////////////////////////////////////////////////////////////////////////////////////
    // @audit - Will call `checkAndPersistMint(...)`; if it reverts, the mint will be unsuccessful.
    ///////////////////////////////////////////////////////////////////////////////////////////////
    gyroConfig.getRootSafetyCheck().checkAndPersistMint(order);
    ...
}
```

```
function dryMint(...) external view override returns (...) {
    ///////////////////////////////////////////////////////////////////////////////////////////////
    // @audit - Will call `isMintSafe(...)`; if an error is returned, the function will end with an unsuccessful mint.
    ///////////////////////////////////////////////////////////////////////////////////////////////
    err = gyroConfig.getRootSafetyCheck().isMintSafe(order);
    if (bytes(err).length > 0) {
        return (0, err);
    }
    ...
}
```

As can be seen, safety checks are done using different functions. There are currently two modules for executing safety checks. These are the `ReserveSafetyManager` and the `VaultSafetyManager`. In the former one, the function `checkAndPersistMint(...)` is a simple wrapper for `isMintSafe(...)`. This is not the case for the `VaultSafetyManager`, in which the function `checkAndPersistMint(...)` executes some extra logic. This extra logic includes handling an error called `OPERATION_SUCCEEDS_BUT_SAFETY_MODE_ACTIVATED`. As the name indicates, this error does not end in an unsuccessful mint. However, when a `dryMint(...)` is executed, the `isMintSafe(...)` function will return this error, and the mint fails. The `checkAndPersistMint(...)` from the `VaultSafetyManager` contract can be seen in the next snippet of code.

```
function checkAndPersistMint(DataTypes.Order memory order) external rootSafetyCheckOnly {
    require(order.mint, Errors.INVALID_ARGUMENT);
    (string memory err, FlowResult[] memory result) = _checkFlows(order);

    if (bytes(err).length > 0) {
        //////////////////////////////////////////////////////////////////////////////////////////
        // @audit - This error is not handled in the case of a `dryMint(...)`
        //////////////////////////////////////////////////////////////////////////////////////////
        if (err.compareStrings(Errors.OPERATION_SUCCEEDS_BUT_SAFETY_MODE_ACTIVATED)) {
            emit SafetyStatus(err);
        } else {
            revert(err);
        }
    }
    _updateFlows(order, result);
}
```

Divergences in the behavior of the `mint(...)` and `dryMint(...)` functions can confuse users.

Recommendation(s): Consider unifying how `mint(...)` and `dryMint(...)` functions work under scenarios that would activate the safety mode.

Status: Fixed.

Update from the client: Fixed in commit 7984e27b5e504157560e6f27509732e2417df32e.

6.8 [Info] Dead code in `VaultRegistry`

File(s): `VaultRegistry.sol`

Description: The function `setInitialPrice` in `VaultRegistry` can be called only by `ReserveManager`. However, the interface `IReserveManager` and the contract `ReserveManager` don't implement a function to `setInitialPrice`.

```
function setInitialPrice(address vault, uint256 initialPrice) external reserveManagerOnly {
    require(vaultAddresses.contains(vault), Errors.VAULT_NOT_FOUND);
    require(vaultsMetadata[vault].priceAtCalibration == 0, Errors.INVALID_ARGUMENT);
    vaultsMetadata[vault].priceAtCalibration = initialPrice;
}
```

Recommendation(s): Consider adding a function to set the initial price in the interface `IReserveManager` and implement it in the contract `ReserveManager` where `governanceOnly` can call.

Status: Fixed.

Update from the client: Fixed in f456f0bdc2c6b5ce2715369364597a17b92c06a1.

6.9 [Info] Excessive memory allocation in the function `batchRelativePriceCheck(...)`

File(s): `CheckedPriceOracle.sol`

Description: The function `batchRelativePriceCheck(...)` allocates redundant memory for the array `priceLevelTwaps`. The maximum number of values that need to be allocated equals the `tokenAddresses.length` because the inner loop breaks once a valid value is found and added to the `priceLevelTwaps` array.

```
function batchRelativePriceCheck(address[] memory tokenAddresses, uint256[] memory prices) internal view returns
    (uint256[] memory)
{
    ///////////////////////////////////////////////////////////////////////////////
    // @audit Maximum length is just tokenAddresses.length()
    ///////////////////////////////////////////////////////////////////////////////
    uint256[] memory priceLevelTwaps = new uint256[](
        tokenAddresses.length * quoteAssetsForPriceLevelTWAPS.length()
    );

    uint256 k;
    for (uint256 i = 0; i < tokenAddresses.length; i++) {
        bool couldCheck = false;

        for (uint256 j = 0; j < assetsForRelativePriceCheck.length(); j++) {
            ...
            couldCheck = true;
            break; // @audit Break when a valid value is added
        }
        ...
    }
    ...
}
```

Recommendation(s): Consider allocating only the necessary memory for the `priceLevelTwaps` array.

Status: Fixed.

Update from the client: Fixed in commit 0f5d9de3b189eebdadb58565f3e0b518b3d897f3.

6.10 [Info] Region detection does not exclude equality cases

File(s): PrimaryAMMV1.sol

Description: The function `isInSecondRegionHigh(...)` returns false if `(derived.baThresholdIIHL >= derived.baThresholdRegionI)`. However, according to Algorithm 2 in the paper, the function returns `false` when `derived.baThresholdIIHL > derived.baThresholdRegionI`, except in the equality case.

```
function isInSecondRegionHigh( State memory normalizedState, uint256 alphaBar, DerivedParams memory derived ) internal
→ pure returns (bool) {
    ////////////////////////////////////////////////////////////////////////////////////////////
    // @audit According to Algorithm 2, it should be derived.baThresholdIIHL > derived.baThresholdRegionI
    ////////////////////////////////////////////////////////////////////////////////////////////
    if (derived.baThresholdIIHL >= derived.baThresholdRegionI) return false;
    if (derived.baThresholdIIHL <= derived.baThresholdRegionII) return true;
...
}
```

Similarly, the same case occurs in the function `isInThirdRegionHigh(...)`. The function should return `false` when `derived.baThresholdIIIHL > derived.baThresholdRegionII`, except in the equality case.

```
function isInThirdRegionHigh(
    State memory normalizedState,
    Params memory params,
    DerivedParams memory derived
) internal pure returns (bool) {
    ////////////////////////////////////////////////////////////////////////////////
    // @audit According to Algorithm 2, it should be
    // derived.baThresholdIIIHL > derived.baThresholdRegionII
    ////////////////////////////////////////////////////////////////////////////////
    if (derived.baThresholdIIIHL >= derived.baThresholdRegionII) return false;
   ...
}
```

Recommendation(s): Consider following the definitions stated in the paper.

Status: Fixed.

Update from the client: We acknowledge that the code does not formally match the paper in the edge cases where `baThresholdIIHL == baThresholdRegionI` and `baThresholdIIIHL == baThresholdRegionII`. However, this does not affect the correctness of the algorithm because in this case, either of `true` or `false` would be appropriate results since we are in fact both in region `h` and `l` (or `H` and `L`, for `isInThirdRegionHigh()`). The behavior of the rest of the algorithm (e.g., the reconstructed b_a and computed redemption amount) will therefore be the same. Nevertheless, for clarity, we believe that the code should match the paper directly. We have adjusted the code in this way. Fixed in 96d499646443e59e4601078a3a19523e0ea8cb0c

6.11 [Info] Return value of 0 from `ecrecover` is not checked

File(s): TrustedSignerPriceOracle.sol

Description: The current implementation of the `TrustedSignerPriceOracle` contract does not check the return value of `ecrecover`, which can be an empty (0x0) address when the signature is invalid. This means that if the contract is set up with a trusted signer equal to the zero address, any submitted price will be accepted, potentially compromising the integrity of the oracle.

Recommendation(s): Consider using the `ECDSA` library from OpenZeppelin to mitigate the issue and adding checks in the constructor to ensure `trustedPriceSigner` is not a zero address.

Status: Fixed.

Update from the client: Fixed in commit 4196e4e3be81e4686a3a88dcaea5a8bffb98a1f4.

6.12 [Info] Users can execute calls from the Motherboard contract

File(s): `Motherboard.sol`

Description: The `Motherboard` contract permits users to execute certain actions before the minting of `GYD`, as shown in the following snippet of code:

```
function mint(
    DataTypes.MintAsset[] calldata assets,
    uint256 minReceivedAmount,
    ExternalAction[] calldata actions
) public returns (uint256 mintedGYDAmount) {
    for (uint256 i = 0; i < actions.length; i++) {
        require(
            externalCallWhitelist.contains(actions[i].target),
            Errors.FORBIDDEN_EXTERNAL_ACTION
        );
        actions[i].target.functionCall(actions[i].data, Errors.EXTERNAL_ACTION_FAILED);
    }

    return mint(assets, minReceivedAmount);
}
```

The `Motherboard` contract holds a significant position within the protocol, as it is authorized to perform key operations such as minting `GYD`. Users may also frequently assign allowances to this contract to enable mint and redeem operations. The functionality allowing external actions to be executed before minting could be abused to take advantage of these situations. A whitelist of approved addresses is available to mitigate these risks, permitting these external actions to call only designated addresses. However, this whitelist approach requires the Gyroscope team to rigorously scrutinize any address before adding it to the whitelist.

Recommendation(s): Every address in the set needs to be carefully analyzed. Other actions that could be taken to eliminate this risk are: a) Remove the external calls feature; b) Use another contract with no privileged role to execute these external calls.

Status: Fixed.

Update from the client: Fixed in 2f5df68cffd7cacb555b5efd07e85b1c8e7effc9.

6.13 [Info] Users might lose everything if they do not call the function `withdraw(...)` when a pending withdrawal is available immediately

File(s): `GydRecovery.sol`

Description: In GydRecovery, when users want to withdraw their stake, they first have to call the function `initiateWithdrawal(...)` and then wait for a certain amount of time before they can actually withdraw to their wallets. This delay ensures that the recovery module can burn the GYD of stakers if there is a burn during the withdrawal wait period. However, after the withdrawal wait period is passed, if users do not call the function `withdraw(...)` immediately, users' withdrawal requests are still affected by burn events. As a result, users could lose their funds if a burn event happens after the waiting period.

```
function withdraw(uint256 withdrawalId) external returns (uint256 amount) {
    PendingWithdrawal memory pending = pendingWithdrawals[withdrawalId];
    require(pending.to == msg.sender, "matching withdrawal does not exist");
    require(pending.withdrawableAt <= block.timestamp, "not yet withdrawable");

    //////////////////////////////////////////////////////////////////
    // @audit If users withdraw a bit late, they could lose all
    //////////////////////////////////////////////////////////////////
    if (pending.createdFullBurnId < nextFullBurnId) {
        delete pendingWithdrawals[withdrawalId];
        userPendingWithdrawalIds[pending.to].remove(withdrawalId);
        emit WithdrawalCompleted(withdrawalId, pending.to, 0, 0);
        return 0;
    }

    positions[pending.to].adjustedAmount -= pending.adjustedAmount;

    amount = pending.adjustedAmount.mulDown(adjustmentFactor);
    gydToken.safeTransfer(pending.to, amount);

    delete pendingWithdrawals[withdrawalId];
    userPendingWithdrawalIds[pending.to].remove(withdrawalId);

    emit WithdrawalCompleted(withdrawalId, pending.to, pending.adjustedAmount, amount);
}
```

Recommendation(s): Consider reviewing the withdrawal mechanism. If a burn occurs after the withdrawal request is available, the available withdrawal requests should not be affected.

Status: Acknowledged.

Update from the client: Risk is accepted. This will be documented to the users.

6.14 [Info] XL value in the code and paper is different for `isInSecondRegion(...)`

File(s): `PrimaryAMMV1.sol`

Description: The function `computeReserveValueRegion(...)` checks if it is in the second region by calling `isInSecondRegion(...)`:

```
function isInSecondRegion(
        State memory normalizedState,
        uint256 alphaBar,
        DerivedParams memory derived
    ) internal pure returns (bool) {
        return
            normalizedState.reserveValue >=
            computeReserveFixedParams(
                normalizedState.redemptionLevel,
                derived.baThresholdRegionII,
                ONE,
                alphaBar,
                0,
                derived.xlThresholdAtThresholdII // @audit this value should not be 1 always?
            );
    }
```

The parameter `derived.xlThresholdAtThresholdII` is related to `XL=1` parameter in Algorithm 2 (line 10), that in this case is expected to be 1. However, `derived.xlThresholdAtThresholdII` is calculated in the function `createDerivedParams(...)` :

```
derived.xlThresholdAtThresholdII = computeXl(
            derived.baThresholdRegionII,
            ONE,
            params.alphaBar,
            0
        );
```

The problem is that the `b(x)` (proposition 2) defined by the algorithm can be different when they use `XL=xlThresholdAtThresholdII` and `XL=1`.

Recommendation(s): Consider updating the code or the paper to the correct parameter.

Status: Fixed.

Update from the client: This is a typo in the paper: in Algorithm 2 Line 10, it should be $x_L = x_L^{II/III}$ instead of $x_L = 1$. (in fact, in the current version of the paper, $x_L^{II/III}$ is computed in Algorithm 1 but never used later on, which is clearly a mistake). We will adjust the paper to use $x_L = x_L^{II/III}$ in Algorithm 2 Line 10. The updated white paper can be found here.

6.15 [Best Practice] Interface `AggregatorV2V3Interface` has multiple functions from deprecated Chainlink API

File(s): `ChainlinkAggregator.sol`

Description: The interface `AggregatorV2V3Interface` contains mutiple functions from a Chainlink's deprecated API: `latestAnswer()`, `latestTimestamp()`, `getTimestamp()`, and `getAnswer()`. Such functions might suddenly stop working in case Chainlink stops supporting deprecated APIs. Chainlink's documentation emphasizes not using them.

Recommendation(s): Consider removing them from the interface to avoid future usage or add comments to warn developers.

Status: Fixed.

Update from the client: Fixed in e8605e621aaa4d7532ec9456b01d86c35a953fa8.

6.16 [Best Practice] Returning the named returns is redundant

File(s): BaseVault.sol

Description: The function depositFor(...) in the BaseVault contract includes a redundant return statement for the named return variable vaultTokensMinted. This is unnecessary since the Solidity compiler automatically handles named returns. The relevant code snippet is provided below:

```
function depositFor(
    address beneficiary,
    uint256 underlyingAmount,
    uint256 minVaultTokensOut
) public override returns (uint256 vaultTokensMinted) {
    uint256 rate = _exchangeRate(true);

    IERC20(underlying).safeTransferFrom(msg.sender, address(this), underlyingAmount);

    vaultTokensMinted = underlyingAmount.divDown(rate);
    require(vaultTokensMinted >= minVaultTokensOut, Errors.TOO_MUCH_SLIPPAGE);

    _mint(beneficiary, vaultTokensMinted);

    /////////////////////////////////////////////////////////
    // @audit No need to return the named variable
    /////////////////////////////////////////////////////////
    return vaultTokensMinted;
}
```

Recommendation(s): Consider removing the unnecessary return statement.

Status: Fixed.

Update from the client: Fixed in caed7733a8cad6955d591a3b81ab75b03cd603ee.

6.17 [Best Practice] Upgradeability issues

File(s): contracts/*

Description: Some contracts in the Gyroscope protocol are upgradeable. However, the best practices are not kept, which may lead to issues during upgrading contracts. Below we list a few points of concern:

1. Inherited contracts do not introduce __gap - The inherited upgradeable contracts should introduce a gap, which acts as a placeholder for new state variables that can be potentially added in the future. This technique protects from storage layout shifts. More information may be found here and here. An example of such a contract is GovernableBase.

2. Implementation contracts don't have disabled initializers - The contracts used as logic contracts for a proxy can be initialized in their context by any address, which may result in unwanted and unexpected behavior. Common mitigation is to invoke _disableInitializers(...) in the constructor to prohibit initialization of the contract by a malicious attacker. More information may be found in the description of the Initializable contract here.

Recommendation(s): Consider following the best practices of upgradeable contracts to avoid possible errors in the upgrade process and maintenance.

Status: Fixed.

Update from the client: Fixed in commit 387c8405da88e3f37696c605a7fab109f5061be0.

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 plays a critical role 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.

The `Gyroscope` team has provided extensive documentation on their protocols. Including formal papers and developer friendly documentation. Their README explains the multiple mechanisms they execute for ensuring safeness of the reserves and vault calibration. Most of this documentation can be found in https://docs.gyro.finance/.

Use of code comments is also present, explaining the goal and behavior of multiple functions. Moreover, the Gyroscope team was available to address any inquiries or concerns raised by the Nethermind auditors.

7.1 Nuance between the function `isInSecondRegionHigh(...)` and the Algorithm in the paper

File(s): `PrimaryAMMV1.sol`

Description: The function `isInSecondRegionHigh(...)` checks if we are in region II high. If `isInSecondRegionHigh(...)` returns false, then it is assumed to be in the second region low. Algorithm 2 defined in the paper describes a condition to check if we are in region II high. We can visualize the condition as a simple expression:

$A \vee (B \wedge C)$

Where,

- A = `derived.baThresholdIIHL <= derived.baThresholdRegionII`
- B = `derived.baThresholdIIHL <= derived.baThresholdRegionI`
- C = `normalizedState.reserveValue >= computeReserveFixedParams(...)`

We are in region II high if the disjunction is `true`. However, when we look in the function `isInSecondRegionHigh(...)`, if A is `true` and B is `false`, the function returns `false` instead returning `true` because B is checked before A.

```
function isInSecondRegionHigh(
    State memory normalizedState,
    uint256 alphaBar,
    DerivedParams memory derived
) internal pure returns (bool) {

    if (derived.baThresholdIIHL >= derived.baThresholdRegionI) return false;
    if (derived.baThresholdIIHL <= derived.baThresholdRegionII) return true;

    return
        normalizedState.reserveValue >=
        computeReserveFixedParams(
            normalizedState.redemptionLevel,
            derived.baThresholdIIHL,
            ONE,
            alphaBar,
            derived.xuThresholdIIHL,
            ONE
        );
}
```

However, this can never happen since `baThresholdRegionI` is always greater or equal to `baThresholdRegionII`. The Gyro team clarified this case according to the math properties definition. The full answer provided by Gyro team is presented below:

> This code is correct due to a mathematical property that may not be obvious. Specifically, the potential counterexample (A is true and B is false) cannot occur because this would mean `baThresholdIIHL <= baThresholdRegionII` and `baThresholdIIHL > derived.baThresholdRegionI`, which implies `baThresholdRegionI < baThresholdRegionII` (in the notation in the paper: $b_a^{I/II} < b_a^{II/III}$). But this is impossible: due to monotonicity of the parameters (α, x_U) as functions of b_a and definition of the regions, we must always have $b_a^{I/II} \geq b_a^{II/III}$.
>
> (Specifically, when we decrease b_a starting at $1 - \varepsilon$, x_U and α will first stay fixed, then α stays fixed and x_U will decrease, then $x_U = 0$ and α will increase. This is by the way that x_U and α are chosen and takes use through regions I, II, III, in order.)
>
> While the code is correct, the reason why it is correct is not obvious and this can be confusing to most readers. We will add a remark in the respective section of the whitepaper and we will refactor the code into the form proposed by you above to more directly match the formula in the paper.

8 Test Suite Evaluation

8.1 Tests Output

```
> brownie test -m 'not mainnetFork and not hypothesis and not endToEnd' --failfast --hypothesis-seed 42
Brownie v1.19.2 - Python development framework for Ethereum

============================= test session starts ==============================
platform linux -- Python 3.10.8, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/runner/work/protocol/protocol, configfile: tests/pytest.ini
plugins: eth-brownie-1.19.2, xdist-1.34.0, hypothesis-6.27.3, web3-5.31.1, forked-1.4.0
collected 269 items / 56 deselected / 213 selected

Launching 'ganache-cli --chain.vmErrorsOnRPCResponse true --server.port 8545 --miner.blockGasLimit 12000000
↪  --wallet.totalAccounts 10 --hardfork london --wallet.mnemonic brownie'...

tests/test_account_setup.py .........                                         [  4%]
tests/test_config.py ......                                                   [  7%]
tests/test_gyd_recovery.py ..........                                         [ 11%]
tests/test_mock_balancer_pool.py ...                                          [ 13%]
tests/test_mock_balancer_vault.py .                                           [ 13%]
tests/test_motherboard.py .......................                             [ 24%]
tests/test_pamm.py .ss...........ssssssssssssssssssssss..........             [ 46%]
tests/test_reserve_manager.py ...                                             [ 47%]
tests/test_stewardship_incentives.py ....                                     [ 49%]
tests/test_vault_registry.py ....                                             [ 51%]
tests/test_vault_safety_mode.py .........                                     [ 55%]
tests/auth/test_cap_authentication.py ..                                      [ 56%]
tests/auth/test_governable.py .....                                           [ 59%]
tests/auth/test_multi_ownable.py ...                                          [ 60%]
tests/fee_handlers/test_static_percentage_fee_handler.py ...                  [ 61%]
tests/oracles/test_asset_registry.py ..........                               [ 67%]
tests/oracles/test_batch_vault_price_oracle.py .                              [ 67%]
tests/oracles/test_chainlink_price_oracle.py .......                          [ 70%]
tests/oracles/test_checked_price_oracle.py ............                       [ 76%]
tests/oracles/test_crash_protected_chainlink_oracle.py ....                   [ 78%]
tests/oracles/test_trusted_signer_price_oracle.py ........                    [ 82%]
tests/reserve/test_reserve_safety_manager.py ...............                  [ 90%]
tests/vaults/test_base_vault.py .....................                         [100%]
> brownie test -m 'not mainnetFork and hypothesis and not endToEnd' --failfast --hypothesis-seed 42
Brownie v1.19.2 - Python development framework for Ethereum

============================= test session starts ==============================
platform linux -- Python 3.10.8, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/runner/work/protocol/protocol, configfile: tests/pytest.ini
plugins: eth-brownie-1.19.2, xdist-1.34.0, hypothesis-6.27.3, web3-5.31.1, forked-1.4.0
collected 269 items / 219 deselected / 50 selected

Launching 'ganache-cli --chain.vmErrorsOnRPCResponse true --server.port 8545 --miner.blockGasLimit 12000000
↪  --wallet.totalAccounts 10 --hardfork london --wallet.mnemonic brownie'...

tests/test_array.py ..                                                        [  4%]
tests/test_fixed_point.py .                                                   [  6%]
tests/test_pamm.py .                                                          [  8%]
tests/oracles/test_checked_price_oracle.py ..                                 [ 12%]
tests/oracles/test_lp_share_pricing.py ..........                             [ 34%]
tests/oracles/test_lp_share_pricing_formulas.py s.......                      [ 50%]
tests/oracles/test_lp_share_pricing_high_prec.py .........s.                  [ 72%]
tests/reserve/test_reserve_safety_manager.py .............                    [100%]
```

```
> brownie test -m mainnetFork --network mainnet-fork --failfast --hypothesis-seed 42
Brownie v1.19.2 - Python development framework for Ethereum

============================= test session starts ==============================
platform linux -- Python 3.10.8, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/runner/work/protocol/protocol, configfile: tests/pytest.ini
plugins: eth-brownie-1.19.2, xdist-1.34.0, hypothesis-6.27.3, web3-5.31.1, forked-1.4.0
collected 269 items / 265 deselected / 4 selected

Launching 'ganache-cli --chain.vmErrorsOnRPCResponse true --server.port 8545 --miner.blockGasLimit 12000000
↪  --wallet.totalAccounts 10 --hardfork london --wallet.mnemonic brownie --fork.url https://mainnet.infura.io/v3/***
↪  --chain.chainId 1'...

tests/oracles/test_chainlink_price_oracle.py .                          [ 25%]
tests/oracles/test_checked_price_oracle.py .                            [ 50%]
tests/oracles/test_crash_protected_chainlink_oracle.py .                [ 75%]
tests/oracles/test_tellor_price_oracle.py .                             [100%]
```

8.2 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

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 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.

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.