Security Review Report
NM-0699-0786 - Alchemix - Strategies

NETHERMIND
SECURITY

(Febuary 2, 2026)

NETHERMIND DRAFT

Contents

NETHERMIND DRAFT

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for Alchemix's **MYT vault system and associated strategy contracts**.

The MYT system is a yield aggregation protocol built on Morpho's VaultV2 infrastructure, enabling passive yield generation for depositors through asset allocation across multiple DeFi protocols. The architecture follows a modular adapter-based design where each strategy contract encapsulates the specific logic required to interact with a target yield-generating protocol.

Alchemix plans to deploy two vaults, one denominated in WETH and one in USDC, across Ethereum Mainnet, Arbitrum, and Optimism. The review covers strategy implementations integrating with diverse protocols including Aave V3, Euler, Lido (wstETH), Autofinance (Tokemak), Peapods, Fluid, and Moonwell.

The audit comprises 1406 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 32 points of attention, where two are classified as `Critical`, six are classified as `High`, six are classified as `Medium`, seven are classified as `Low` and eleven are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

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

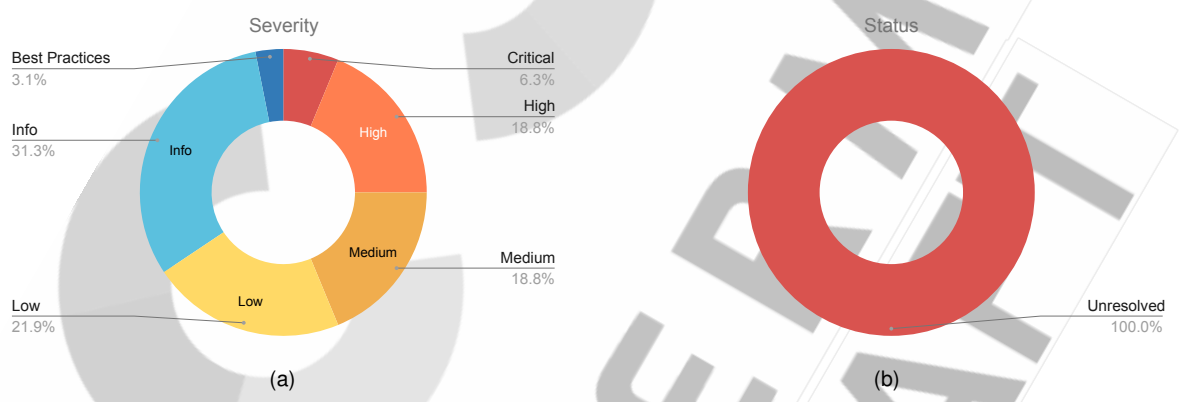


Fig. 1: Distribution of issues: Critical (2), **High** (6), **Medium** (6), **Low** (7), **Undetermined** (0), **Informational** (10), **Best Practices** (1). **Distribution of status: Fixed** (0), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (32)

Summary of the Audit

Audit Type	Security Review
Initial Report	Febuary 2, 2026
Final Report	-
Initial Commit	`alchemix-finance/myt-ac:` `555e26f8` `alchemix-finance/myt :` `9d527998`
Intermediary Commit	`alchemix-finance/myt-ac:` `deedf517` `alchemix-finance/myt :` `8a6b3d63`
Final Commit	`alchemix-finance/myt-ac:` `alchemix-finance/myt :`
Documentation Assessment	Medium
Test Suite Assessment	Low

NETHERMIND DRAFT

2 Audited Files

2.1 Access control repository (alchemix-finance/myt-ac)

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/AlchemistCurator.sol	139	7	5.0%	32	178
2	src/AlchemistAllocator.sol	50	22	44.0%	13	85
3	src/utils/PermissionedProxy.sol	58	1	1.7%	14	73
4	src/interfaces/IAllocator.sol	5	5	100.0%	2	12
5	src/interfaces/IAlchemistCurator.sol	24	1	4.2%	5	30
	Total	**276**	**36**	**13.0%**	**66**	**378**

2.2 Strategies repository (alchemix-finance/myt)

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/MYTStrategy.sol	201	111	55.2%	56	368
2	src/AlchemistGate.sol	9	1	11.1%	2	12
3	src/strategies/WstethMainnet.sol	68	12	17.6%	20	100
4	src/strategies/optimism/AaveV3OPUSDCStrategy.sol	44	9	20.5%	10	63
5	src/strategies/optimism/MoonwellUSDCStrategy.sol	75	18	24.0%	15	108
6	src/strategies/optimism/MoonwellWETHStrategy.sol	76	17	22.4%	17	110
7	src/strategies/arbitrum/FluidARBUSDCStrategy.sol	38	4	10.5%	8	50
8	src/strategies/arbitrum/EulerARBWETHStrategy.sol	38	4	10.5%	8	50
9	src/strategies/arbitrum/EulerARBUSDCStrategy.sol	38	4	10.5%	8	50
10	src/strategies/arbitrum/AaveV3ARBWETHStrategy.sol	50	9	18.0%	12	71
11	src/strategies/arbitrum/AaveV3ARBUSDCStrategy.sol	50	9	18.0%	9	68
12	src/strategies/mainnet/PeapodsETH.sol	47	5	10.6%	12	64
13	src/strategies/mainnet/EulerUSDCStrategy.sol	38	4	10.5%	8	50
14	src/strategies/mainnet/EulerWETHStrategy.sol	38	4	10.5%	8	50
15	src/strategies/mainnet/PeapodsUSDC.sol	38	0	0.0%	8	46
16	src/strategies/mainnet/TokeAutoUSDStrategy.sol	107	14	13.1%	22	143
17	src/strategies/mainnet/MorphoYearnOGWETH.sol	52	21	40.4%	11	84
18	src/strategies/mainnet/TokeAutoEth.sol	123	40	32.5%	27	190
	Total	**1130**	**286**	**25.3%**	**261**	**1677**

NETHERMIND DRAFT

3 Summary of Issues

	Finding	Severity	Update
1	Allocation and deallocation from strategies revert due to inconsistent data decoding	Critical	Unresolved
2	Incorrect stETH denomination in `WstethMainnetStrategy`'s `_allocate(...)` causes funds to get stuck and MYT share price to decrease	Critical	Unresolved
3	Claiming rewards in Tokemak strategies reverts due to incorrect recipient address	High	Unresolved
4	Deallocation from `wstETH` strategy reverts due to missing vault approval for pulling assets	High	Unresolved
5	Incorrect slippage check logic in the `dexSwap(...)` function causes reverts during allocate and deallocate flows	High	Unresolved
6	Missing access control in `submitSetAllocator(...)`	High	Unresolved
7	Permissionless `forceDeallocate(...)` combined with missing slippage protection enables sandwich attacks on strategy swaps	High	Unresolved
8	The `claimRewards(...)` function is permissionless and allows arbitrary swap data, leading to theft of rewards	High	Unresolved
9	Deallocation unwraps excess `wstETH`, leaving unaccounted WETH in the strategy	Medium	Unresolved
10	Inconsistent Aave pool address handling may lead to lost funds	Medium	Unresolved
11	Inconsistent `killSwitch` logic blocks pending withdrawals	Medium	Unresolved
12	The `_allocate(...)` function causes valid deposits to revert due to incorrect slippage check	Medium	Unresolved
13	Unhandled TOKE rewards due to incorrect reward handling during deallocation	Medium	Unresolved
14	Unhandled Tokemak reward staking flow may break reward claiming logic	Medium	Unresolved
15	Aave V3 Strategies do not implement rewards claiming from `RewardsController`	Low	Unresolved
16	Incorrect unit conversion in `_previewAdjustedWithdraw(...)` leads to inaccurate asset estimates	Low	Unresolved
17	Inflated asset estimation in `_previewAdjustedWithdraw`	Low	Unresolved
18	Moonwell strategies do not handle incentive rewards	Low	Unresolved
19	Morpho V2 vault `maxRate` and other parameters are not set	Low	Unresolved
20	The `WstethMainnetStrategy` does not validate that the `dexSwap(...)` output meets the requested `amount`	Low	Unresolved
21	The `dexSwap(...)` function fails to return the swapped amount resulting in incorrect reward accounting	Low	Unresolved
22	Absence of slippage check in `dexSwap` due to unused `minAmountOut` parameter	Info	Unresolved
23	Deployment script for Fluid strategy uses wrong address	Info	Unresolved
24	Strategies may not account for entire native asset amount	Info	Unresolved
25	The `proxy(...)` function allows operators to bypass access controls in `AlchemistCurator`	Info	Unresolved
26	TokeAuto strategies optimistically preview withdrawals	Info	Unresolved
27	Unnecessary timelock pattern for cap decrease functions in `AlchemistCurator`	Info	Unresolved
28	Unsupported action types silently return 0 instead of reverting	Info	Unresolved
29	Unused `minFinalOut` parameter in `MYTStrategy`	Info	Unresolved
30	`AlchemistCurator` doesn't support all curator functions	Info	Unresolved
31	`slippageBPS` cannot be updated after deployment	Info	Unresolved
32	Unnecessary interface declarations	Best Practices	Unresolved

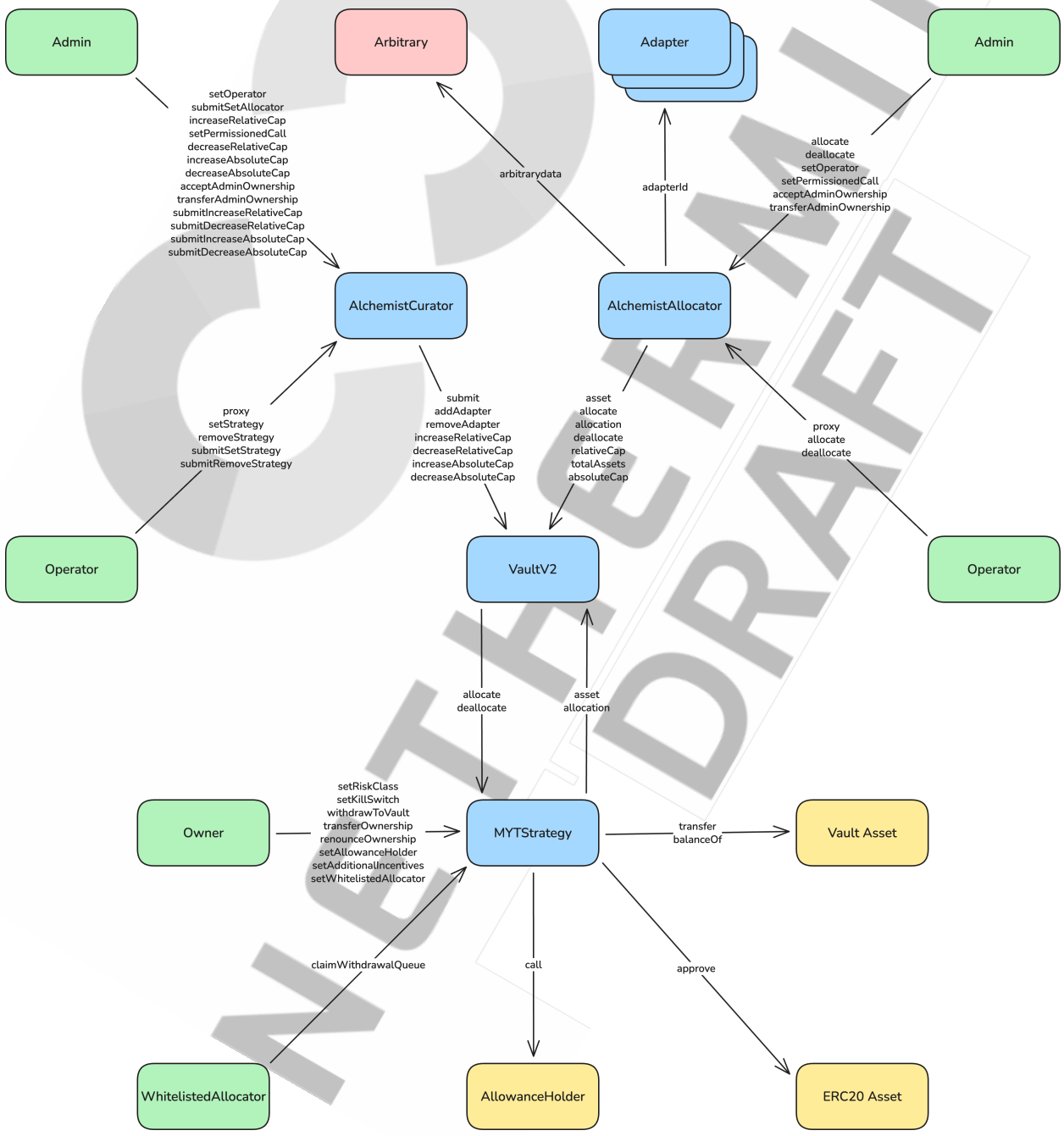
NETHERMIND DRAFT

4 System Overview

The Alchemix MYT protocol is a yield aggregation system built on the Morpho V2 Vault infrastructure. The protocol enables users to deposit assets into a vault that allocates funds across multiple yield-generating strategies, maximizing returns while managing risk through a cap and risk classification system. The protocol is composed of the following core components:

- **MYT Vault**: The central ERC-4626 compliant vault that serves as the liquidity hub. It manages user deposits, tracks total assets, and orchestrates the movement of funds into various adapters.
- **MYT Strategies**: Modular, protocol-specific adapters that connect the MYT Vault to external DeFi protocols. Each strategy encapsulates the logic for depositing, withdrawing, and claiming rewards from its specific target.
- **Alchemist Allocator**: A permissioned allocation manager that has the allocator role in the vault. It validates all allocation requests against strict risk controls and cap limits before interacting with the vault.
- **Alchemist Curator**: A governance and configuration manager responsible for onboarding strategies and adjusting allocation caps.
- **Risk Classifier**: The `AlchemistStrategyClassifier` contract categorizes strategies into specific risk levels (Low, Medium, High). It enforces global and local allocation caps based on these risk profiles.

The figure below presents a high-level view of the system architecture, highlighting the main components and their interactions.



NETHERMIND DRAFT

4.1 Vault Funds Management

4.1.1 Allocation Execution Flow

To allocate funds to a specific strategy, the flow begins by calling one of the allocation functions on the `AlchemistAllocator` contract. The allocator validates the request against the calculated effective limit, defined as the minimum of absolute, relative, and risk-based caps.

Once validated, the allocator encodes calldata specifying the action type and any distinct swap parameters. It then calls the vault's `allocate` function, which transfers assets to the adapter and triggers the strategy's internal allocation logic.

```
// AlchemistAllocator.sol
function allocate(address adapter, uint256 amount) external;
function allocateWithSwap(address adapter, uint256 amount, bytes memory txData) external;

// VaultV2.sol
function allocate(address adapter, bytes memory data, uint256 assets) external;

// MYTStrategy.sol
function allocate(bytes memory data, uint256 assets, bytes4 selector, address sender) external onlyVault returns
↪  (bytes32[] memory strategyIds, int256 change);
```

Upon receiving the call, the strategy executes the logic corresponding to the specified action type. For direct actions, it interacts directly with the underlying protocol. For swap actions, it utilizes 0x calldata to perform a DEX swap before depositing.

Finally, the strategy returns the change in allocation value to the vault, which updates its internal allocation mapping.

4.1.2 Deallocation Execution Flow

Deallocation mirrors the allocation process but operates in reverse. The process initiates with the allocator calling the vault, which triggers the strategy's `deallocate` function.

The strategy withdraws assets from the underlying protocol, performing swaps or unwrapping operations if required. Crucially, the strategy must approve the vault to move the withdrawn assets. The vault then pulls the specified amount using `transferFrom` and updates its internal allocation tracking based on the change returned by the strategy.

```
// AlchemistAllocator.sol
function deallocate(address adapter, uint256 amount) external;
function deallocateWithSwap(address adapter, uint256 amount, bytes memory txData) external;

// VaultV2.sol
function deallocate(address adapter, bytes memory data, uint256 assets) external;

// MYTStrategy.sol
function deallocate(bytes memory data, uint256 assets, bytes4 selector, address sender) external onlyVault returns
↪  (bytes32[] memory strategyIds, int256 change);
```

4.1.3 Action Types

The allocator defines three action types to accommodate various integration requirements:

- **Direct**: Used for direct allocation or deallocation interactions with the underlying protocol, such as deposits or withdrawals from an ERC-4626 vault.
- **Swap**: Used when a token swap is required to enter or exit a position. Swaps are executed through the 0x protocol, with the encoded quote supplied by the allocator.
- **UnwrapAndSwap**: Used during deallocations that require unwrapping a token before swapping (e.g., unwrapping wstETH to stETH, then swapping to WETH).

4.2 Vault Valuation and Interest Accrual

The vault's total value is derived through the `totalAssets()` function, which is critical for interest accrual. The total assets are calculated as the sum of idle assets (the vault's direct token balance) and the value of all adapter positions reported via `realAssets()`.

As a result, it is essential that each strategy correctly reports its current underlying value through its `realAssets()` implementation, which is tailored to the accounting mechanics of the underlying protocol.

NETHERMIND DRAFT

4.3 Risk Classifier

The `AlchemistStrategyClassifier` contract categorizes strategies by risk level and enforces exposure caps accordingly. Each strategy is assigned a risk level (LOW, MEDIUM, or HIGH), which determines its allocation constraints.

The classifier defines two types of risk-based caps:

- **Global Cap**: The maximum aggregate value that can be allocated across all strategies within a given risk level.
- **Local Cap**: The maximum value that can be allocated to an individual strategy within that risk level.

During allocation validation, the `AlchemistAllocator` queries the classifier to ensure that neither global nor local risk limits are exceeded.

4.4 Alchemist Curator

The Alchemist Curator acts as a permissioned proxy representing the curator role for the MYT Vault. It provides a controlled interface for governance operations, including adding or removing strategies and adjusting allocation caps.

Administrators can manage strategies through functions such as `submitSetStrategy` and `removeStrategy`, as well as update absolute or relative caps.

Most curator actions are subject to a two-step timelock process. The curator must first submit a configuration change via the `submit` function. After the time lock expires, the action may be executed by anyone. Actions that reduce risk, such as lowering caps, may be executed immediately to allow rapid defensive responses.

4.5 MYT Strategies (Vault Adapters)

MYT Strategies function as vault adapters and inherit from the `MYTStrategy` base contract. Each strategy encapsulates the logic required to interact with third-party yield-generating protocols such as Aave, Tokemak, or Lido.

Strategies are chain and protocol specific and are identified by a unique `adapterId` derived from the contract address. They implement the `allocate()` and `deallocate()` functions, which are restricted to calls from the vault.

In addition, strategies are responsible for reporting their underlying value via `realAssets()`, providing yield snapshots, and exposing emergency controls such as a kill switch.

NETHERMIND DRAFT

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.

NETHERMIND DRAFT

6 Issues

6.1 [Critical] Allocation and deallocation from strategies revert due to inconsistent data decoding

File(s): src/MYTStrategy.sol

Description: When allocating funds to a strategy, the admin calls the `AlchemistAllocator.allocate` function. This function forwards the call to the vault's `allocate` function, passing the adapter address, the amount to be allocated, and a `bytes` payload that encodes the adapter's previous allocation value.

```
function allocate(address adapter, uint256 amount) external {
    // ...
    // pass the old allocation to the adapter
    bytes memory oldAllocation = abi.encode(vault.allocation(id));

    vault.allocate(adapter, oldAllocation, amount);
}
```

The vault then invokes the strategy's `allocate` function (`MYTStrategy.allocate`), forwarding the same encoded `data` parameter.

However, the strategy decodes this data inconsistently with how it is encoded by the allocator. Specifically, `MYTStrategy.allocate` expects the `data` parameter to be either empty or ABI-encoded as a `VaultAdapterParams` struct. If `data` is non-empty, it is always decoded as this struct.

Attempting to decode an encoded `uint256` as a `VaultAdapterParams` struct causes the transaction to revert.

```
function allocate(bytes memory data, uint256 assets, bytes4 selector, address sender)
    external
    onlyVault
    returns (bytes32[] memory strategyIds, int256 change)
{
    require(!killSwitch, StrategyAllocationPaused(address(this)));
    require(assets > 0, "Zero amount");
    uint256 amountAllocated;

    if (data.length == 0) {
        amountAllocated = _allocate(assets);
    } else {
        // @audit-issue data is encoded as a uint256 but decoded as
        // a VaultAdapterParams struct
        VaultAdapterParams memory adapterParams = abi.decode(data, (VaultAdapterParams));
        ActionType action = adapterParams.action;

        if (action == ActionType.direct) {
            amountAllocated = _allocate(assets);
        } else if (action == ActionType.swap) {
            amountAllocated = _allocate(assets, adapterParams.swapParams.txData);
        } else {
            revert("Invalid action");
        }
    }
    // ...
}
```

The same inconsistent data encoding and decoding is also present in the deallocation flow, causing deallocations to revert under the same conditions.

Recommendation(s): Consider updating the `AlchemistAllocator` to encode the `data` parameter in a format that is consistent with what the strategy expects (i.e., `VaultAdapterParams` struct).

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.2 [Critical] Incorrect stETH denomination in `WstethMainnetStrategy`'s `_allocate(...)` causes funds to get stuck and MYT share price to decrease

File(s): src/strategies/WstethMainnet.sol

Description: The `WstethMainnetStrategy` contract is responsible for managing deposits into Lido. The `_allocate(...)` function handles the conversion of WETH to ETH, the deposit into Lido to obtain `stETH`, and the subsequent wrapping of `stETH` into `wstETH`.

However, the logic incorrectly handles the return value from the Lido `submit(...)` function. The `submit(...)` function returns the amount of shares minted, not the amount of `stETH` assets (balance) created. The strategy uses this share amount as the input for `wsteth.wrap(...)`, which expects an asset amount.

```
function _allocate(uint256 amount) internal override returns (uint256 depositReturn) {
    // ...
    // @audit-issue The submit(...) function returns shares, not assets.
    uint256 stETHReceived = steth.submit{value: amount}(address(0));
    // ...
    // @audit-issue This wraps the shares amount of stETH instead of the full asset balance,
    // leaving funds stuck.
    uint256 wstETHMinted = wsteth.wrap(stETHReceived);

    return amount;
}
```

Since the `stETH` share price is currently greater than 1 (1 share represents > 1 stETH), the number of shares returned is significantly lower than the actual amount of `stETH` minted.

For example, if `10 ETH` are allocated:

1. `steth.submitvalue: 10e18(...)` is called;
2. Assuming a share price of roughly `1.2`, this returns approximately `8.33e18` shares;
3. The strategy calls `wsteth.wrap(8.33e18)`;
4. Only `8.33e18` stETH are wrapped into `wstETH`;
5. The remaining `1.67e18` stETH are left unwrapped in the strategy contract;

The `_totalValue()` function calculates the strategy's assets based solely on the `wstETH` balance. Because the unwrapped `stETH` is ignored by `_totalValue()`, the Morpho Vault perceives this as an immediate loss of funds. This causes the Vault's `totalAssets()` to decrease and the MYT share price to drop immediately after allocation.

Recommendation(s): Consider updating the logic to correctly determine the amount of `stETH` assets received.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.3 [High] Claiming rewards in Tokemak strategies reverts due to incorrect recipient address

File(s): src/TokeAutoUSDStrategy.sol, src/TokeAutoEthStrategy.sol

Description: The `_claimRewards` function is responsible for claiming TOKE rewards from the Tokemak `Rewarder` contract, swapping the claimed TOKE into the underlying asset, and transferring the resulting tokens to the MYT vault.

However, when invoking `rewarder.getReward`, the MYT vault (`address(MYT)`) is incorrectly specified as the recipient of the rewards. As a result, the claimed TOKE tokens are transferred directly to the vault instead of remaining in the strategy to be swapped later.

```
function _claimRewards(address token, bytes memory quote)
    internal
    override
    returns (uint256 rewardsClaimed)
{
    require(
        token == 0x2e9d63788249371f1DFC918a52f8d799F4a38C94 && quote.length > 0,
        "params"
    );

    rewardsClaimed = rewarder.earned(address(this));
    // @audit rewards are sent to the MYT vault
    rewarder.getReward(address(this), address(MYT), false);

    uint256 amountOut = dexSwap(token, rewardsClaimed, 0, quote);
    TokenUtils.safeTransfer(address(MYT.asset()), address(MYT), amountOut);
}
```

Since the vault's value is accounted for using its underlying asset balance, TOKE tokens transferred directly to the vault are not included in its valuation. Moreover, because the strategy no longer holds the claimed TOKE, the subsequent swap attempt reverts due to insufficient token balance.

As a result, reward claiming for Tokemak strategies consistently fails, and rewards are not properly realized.

Recommendation(s): Consider updating the `recipient` parameter in the `rewarder.getReward` call to point to the strategy itself (`address(this)`) rather than the MYT vault.

Status: Unresolved

Update from the client:

6.4 [High] Deallocation from `wstETH` strategy reverts due to missing vault approval for pulling assets

File(s): src/strategies/WstethMainnet.sol

Description: During the deallocation flow, the vault first calls `adapter.deallocate` to unwind the position. After this call completes, the vault pulls the deallocated funds from the adapter using `safeTransferFrom`. For this mechanism to succeed, each adapter must grant the vault sufficient allowance to transfer the deallocated assets.

However, the `_deallocate` function in the strategy does not approve the MYT vault to pull the deallocated funds. As a result, when the vault attempts to transfer the assets from the strategy using `safeTransferFrom`, the transaction reverts due to insufficient allowance.

```
function _deallocate(uint256 amount, bytes memory callData)
    internal
    override
    returns (uint256)
{
    uint256 stETHBefore = TokenUtils.safeBalanceOf(address(steth), address(this));
    wsteth.unwrap(amount);
    uint256 stETHAfter = TokenUtils.safeBalanceOf(address(steth), address(this));
    uint256 stETHReceived = stETHAfter - stETHBefore;
    // Swap stETH -> WETH via 0x (callData is the 0x tx data for stETH -> WETH swap)
    dexSwap(address(steth), stETHReceived, 0, callData);

    // @audit Missing approval for the MYT vault to pull the assets
    return amount;
}
```

Because no approval is granted, the vault is unable to retrieve the deallocated assets, causing the deallocation process to fail.

Recommendation(s): Consider approving the MYT vault to pull the deallocated assets before returning from `_deallocate`.

Status: Unresolved

Update from the client:

6.5 [High] Incorrect slippage check logic in the `dexSwap(...)` function causes reverts during allocate and deallocate flows

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

Description: The `MYTStrategy` contract includes the `dexSwap(...)` function. This helper function allows strategies to perform token swaps via the 0x Protocol. It is primarily intended for two use cases: swapping reward tokens into the vault's underlying asset, or performing immediate withdrawals via a DEX to bypass long withdrawal queues imposed by third-party protocols.

The function accepts a `minAmountOut` parameter to enforce slippage protection during the swap. However, the logic used to verify the received amount is flawed because it snapshots the `balanceBefore` *after* the external swap call has already been executed.

```
function dexSwap(address token, uint256 amount, uint256 minAmountOut, bytes memory callData) internal returns (uint256)
↪ {
    IERC20(token).approve(allowanceHolder, type(uint256).max);

    // @audit The swap is executed here via a low-level call.
    (bool success, ) = allowanceHolder.call(callData);

    // ...
    // @audit-issue `balanceBefore` is read AFTER the swap has occurred.
    uint256 balanceBefore = IERC20(MYT.asset()).balanceOf(address(this));
    require(success, "0x exception");
    uint256 balanceAfter = IERC20(MYT.asset()).balanceOf(address(this));

    // @audit `balanceAfter` and `balanceBefore` will be identical.
    // If `minAmountOut` > 0, this check will always fail and revert.
    require(balanceAfter - balanceBefore >= minAmountOut, "0xsa");
    IERC20(token).approve(allowanceHolder, 0);
    // ...
}
```

Since both `balanceBefore` and `balanceAfter` reflect the token balance after the swap, the calculated difference is always zero. Consequently, any attempt to use this function with a valid `minAmountOut` (greater than zero) will inevitably revert due to the slippage check require statement. To avoid reverting, the caller must pass `0` as `minAmountOut`, which completely disables slippage protection and exposes the system to sandwich attacks. This issue impacts any strategy relying on `dexSwap` for allocation or deallocation flows.

Recommendation(s): Consider capturing the `balanceBefore` snapshot before the external `call` to the `allowanceHolder`.

Status: Unresolved

Update from the client:

6.6 [High] Missing access control in `submitSetAllocator(...)`

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

Description: The contract `AlchemistCurator` implements functions to manage key parameters on the MYT vault, including absolute caps, relative caps, and strategies. These functions are protected by the `onlyOperator` or `onlyAdmin` modifiers except `submitSetAllocator`, which has no access control. As a result, any user can nominate themselves as an allocator and (mis)manage funds without permission, potentially leading to loss of funds.

Recommendation(s): Consider adding the modifier `onlyAdmin` to the function `submitSetAllocator`.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.7 [High] Permissionless `forceDeallocate(...)` combined with missing slippage protection enables sandwich attacks on strategy swaps

File(s): src/strategies/WstethMainnet.sol

Description: The `VaultV2` contract provides a `forceDeallocate(...)` function that allows any user to trigger deallocation from an adapter, paying a penalty in return. This function accepts arbitrary `bytes memory data` that is passed through to the adapter's `deallocate(...)` function without any validation.

```
function forceDeallocate(address adapter, bytes memory data, uint256 assets, address onBehalf)
    external
    returns (uint256)
{
    // @audit The `data` parameter is passed unvalidated to deallocateInternal.
    bytes32[] memory ids = deallocateInternal(adapter, data, assets);
    // ...
}
```

When the adapter is a strategy that supports swap functionality (such as `WstethMainnetStrategy`), the data is decoded into `VaultAdapterParams`, which includes a `SwapParams` struct containing attacker-controlled `txData`.

```
function deallocate(...) external onlyVault returns (...) {
    VaultAdapterParams memory adapterParams = abi.decode(data, (VaultAdapterParams));
    // ...
    if (action == ActionType.unwrapAndSwap) {
        // @audit Attacker's callData flows to _deallocate.
        amountDeallocated = _deallocate(assets,
        ↪  adapterParams.swapParams.txData,adapterParams.swapParams.minIntermediateOut);
    }
   }
```

In `WstethMainnetStrategy`, the `_deallocate(...)` function unwraps `wstETH` to `stETH` and then calls `dexSwap(...)` with the attacker-controlled `callData` and a hardcoded `minAmountOut` of 0.

```
function _deallocate(uint256 amount, bytes memory callData, uint256 minIntermediateOut)
    internal override returns (uint256)
{
    // ...
    wsteth.unwrap(wstETHToUnwrap);
    // ...
    // @audit minAmountOut is hardcoded to 0.
    dexSwap(address(weth), address(steth), stETHReceived, 0, callData);
    // ...
}
```

The `dexSwap(...)` function in `MYTStrategy` accepts a `minAmountOut` parameter; however, this value is never actually enforced.

```
function dexSwap(address to, address from, uint256 amount, uint256 minAmountOut, bytes memory callData)
    internal returns (uint256)
{
    IERC20(from).approve(allowanceHolder, type(uint256).max);
    uint256 targetBalanceBefore = IERC20(to).balanceOf(address(this));
    (bool success, ) = allowanceHolder.call(callData);
    require(success, "0x exception");
    uint256 targetBalanceAfter = IERC20(to).balanceOf(address(this));
    IERC20(from).approve(allowanceHolder, 0);
    // @audit minAmountOut is never checked against the actual output.
    return targetBalanceAfter - targetBalanceBefore;
}
```

An attacker can exploit this by calling `forceDeallocate(...)` with malicious `callData` that routes the swap through a pool they control or can manipulate. Since there is no slippage protection (`minAmountOut` is 0 and never enforced), the attacker can sandwich the transaction to extract value from the strategy's funds. The `minFinalOut` parameter in `SwapParams` is also never used in the deallocation logic.

Recommendation(s): Consider implementing proper slippage protection by enforcing the `minAmountOut` check in `dexSwap(...)`. The function should verify that the actual swap output meets the minimum threshold. Additionally, consider validating swap parameters against trusted sources rather than accepting arbitrary calldata from untrusted callers.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.8 [High] The `claimRewards(...)` function is permissionless and allows arbitrary swap data, leading to theft of rewards

File(s): src/MYTStrategy.sol, src/strategies/mainnet/TokeAutoEth.sol

Description: Certain MYT strategies, such as the `TokeAutoEthStrategy` (integrating with Auto Finance/Tokemak), accrue additional reward tokens (e.g., TOKE) on top of the standard yield. To process these rewards, the `MYTStrategy` contract exposes a `claimRewards(...)` function. This function claims the rewards from the external protocol, swaps them for the vault's asset using the 0x protocol, and transfers the proceeds to the vault.

The `claimRewards(...)` function in the base strategy contract is `public` and accepts arbitrary `quote` bytes, which are passed to the 0x swap execution.

```
// src/MYTStrategy.sol
function claimRewards(address token, bytes memory quote) public virtual returns (uint256) {
    require(!killSwitch, "emergency");
    // @audit-issue Function is permissionless and passes user-controlled quote to internal logic.
    _claimRewards(token, quote);
}
```

In the implementation within `TokeAutoEthStrategy`, this function executes the swap with no slippage protection. Specifically, the `minAmountOut` parameter passed to `dexSwap` is hardcoded to `0`.

```
// src/strategies/mainnet/TokeAutoEth.sol
function _claimRewards(address token, bytes memory quote)
    internal
    override
    returns (uint256 rewardsClaimed)
{
    // ...
    rewardsClaimed = rewarder.earned(address(this));
    rewarder.getReward(address(this), address(MYT), false);

    // @audit-issue dexSwap is called with 0 as minAmountOut
    // and the quote is user-controlled.
    uint256 amountOut = dexSwap(address(MYT.asset()), token, rewardsClaimed, 0, quote);
    TokenUtils.safeTransfer(address(MYT.asset()), address(MYT), amountOut);
}
```

Because the function is permissionless and does not validate the `quote` data or enforce a minimum output amount, an attacker can call `claimRewards(...)` whenever rewards are accrued. The attacker can provide malicious `quote` data to route the swap through a liquidity pool they control with manipulated exchange rates, or sandwich the transaction. Since the strategy accepts `0` output tokens as valid, the attacker can extract nearly 100% of the reward value.

Recommendation: Consider implementing access control on the `claimRewards(...)` function to ensure that only trusted actors can initiate the reward claiming process and specify the swap parameters.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.9 [Medium] Deallocation unwraps excess wstETH, leaving unaccounted WETH in the strategy

File(s): src/strategies/WstethMainnet.sol

Description: When deallocating funds from the WstethMainnetStrategy, the _deallocate function first unwraps wstETH into stETH and then swaps the received stETH into WETH. After the deallocation completes, the vault pulls the requested amount of WETH, specified by the amount parameter, from the strategy.

However, the function incorrectly treats the amount parameter as an amount of wstETH when calling wsteth.unwrap(amount), even though amount represents the amount of WETH that the vault intends to retrieve.

Because wstETH is a value-accruing token where 1 wstETH represents more than 1 stETH, unwrapping amount of wstETH yields more stETH than is required to obtain amount of WETH. This excess stETH is subsequently swapped into WETH.

Consequently, the surplus WETH produced by the swap remains in the strategy contract and is not transferred to the vault. This will cause the vault's total value to decrease, since not all deallocated value is transferred to the vault.

```
function _deallocate(uint256 amount, bytes memory callData)
    internal
    override
    returns (uint256)
{
    // Unwrap wstETH -> stETH
    uint256 stETHBefore = TokenUtils.safeBalanceOf(address(steth), address(this));
    // @audit Wrongly interprets `amount` as a wstETH amount
    wsteth.unwrap(amount);
    uint256 stETHAfter = TokenUtils.safeBalanceOf(address(steth), address(this));
    uint256 stETHReceived = stETHAfter - stETHBefore;

    // Swap stETH -> WETH via 0x (callData is the 0x tx data for stETH -> WETH swap)
    dexSwap(address(steth), stETHReceived, 0, callData);

    return amount;
}
```

Recommendation(s): Consider computing the precise amount of wstETH required to unwrap in order to obtain the requested amount of WETH, and unwrap only that amount.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.10 [Medium] Inconsistent Aave pool address handling may lead to lost funds

File(s): src/strategies/optimism/AaveV3OPUSDCStrategy.sol, src/strategies/arbitrum/AaveV3ARBUSDCStrategy.sol, src/strategies/arbitrum/AaveV3ARBWETHStrategy.sol

Description: In Aave V3, it is generally recommended to fetch the Pool address via the `PoolAddressesProvider`, as Aave governance may update the Pool address over time. However, the Aave strategies handle the Pool address inconsistently across networks.

The Optimism strategy (`AaveV3OPUSDCStrategy`) hardcodes the Pool and aToken addresses at deployment time:

```
constructor(
    address _myt,
    StrategyParams memory _params,
    address _usdc,
    address _aUSDC,
    address _pool
) MYTStrategy(_myt, _params) {
    usdc = IERC20(_usdc);
    pool = IAavePool(_pool);
    aUSDC = IAaveAToken(_aUSDC);
}
```

In contrast, the Arbitrum strategies dynamically fetch the Pool address from the `PoolAddressesProvider` on every allocation and deallocation:

```
function _allocate(uint256 amount) internal override returns (uint256) {
    IAavePool pool = IAavePool(poolProvider.getPool());
   // ...
}
```

This Arbitrum approach introduces several flaws. When Aave governance deploys a new Pool implementation, new aToken contracts are also deployed for each supported asset. The previous `aUSDC` or `aWETH` contracts are no longer associated with the new Pool.

However, the Arbitrum strategies update the Pool address dynamically while keeping the aToken addresses immutable. Since Pool and aToken addresses are tightly coupled and must be updated together, this design can result in the strategy interacting with a new Pool while referencing outdated aToken contracts, leading to state inconsistencies.

Additionally, dynamically resolving the Pool address on every allocation and deallocation causes the strategy to immediately lose the ability to interact with the previous Pool once an update occurs. If funds remain deposited in the old Pool during a migration, the strategy no longer has a reference to withdraw them. Given that these adapters are not upgradeable, any funds left in the deprecated Pool would become permanently stuck.

Recommendation(s): Consider standardizing Pool address handling across all Aave strategies. One option is to follow the Optimism approach and hardcode both the Pool and aToken addresses at deployment. When Aave governance deprecates a Pool, operators can explicitly exit all positions, remove the strategy, and deploy a new one configured with the updated addresses.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.11 [Medium] Inconsistent `killSwitch` logic blocks pending withdrawals

File(s): src/MYTStrategy.sol

Description: The `MYTStrategy` implements a `killSwitch` boolean intended to pause operations during emergencies, such as a compromise of the underlying protocol. However, the current implementation contains logical inconsistencies that may trap user funds and contradict the documented behavior.

The `killSwitch` permits `deallocate()` (initiation of withdrawals) but reverts on `claimWithdrawalQueue()` (finalization of withdrawals). For underlying protocols utilizing a two-step withdrawal mechanism, this creates a critical deadlock: allocators can request a withdrawal, but they are prevented from finalizing it. This effectively traps funds that are already in the withdrawal pipeline precisely when users most need to exit.

Additionally, the inline documentation states that the `killSwitch` should allow operations to be "simply bypassed without reverts" to prevent external allocators from failing. However, the actual code implementation enforces a hard revert when the switch is active.

```
/// @notice This value is true when the underlying protocol is known to
/// experience issues or security incidents. In this case, the allocation step is simply
/// bypassed without reverts (to keep external allocators from reverting).
bool public killSwitch;
```

Recommendation(s): Consider removing the `killSwitch` restriction on `claimWithdrawalQueue()` to ensure allocators can complete withdrawals and exit the protocol during emergencies. Additionally, resolve the discrepancy between the code and comments. Either update the code to return silently as described or update the documentation to reflect the reverting behavior.

Status: Unresolved

Update from the client:

6.12 [Medium] The `_allocate(...)` function causes valid deposits to revert due to incorrect slippage check

File(s): src/strategies/mainnet/MorphoYearnOGWETHStrategy.sol

Description: The `_allocate(...)` function in the `MorphoYearnOGWETHStrategy` contract performs a slippage check using the `_previewAdjustedWithdraw(...)` function. The function intends to verify that the strategy received enough shares for the deposited amount.

However, the logic incorrectly uses a withdrawal preview function to validate a deposit operation.

```
function _allocate(uint256 amount) internal override returns (uint256) {
    require(TokenUtils.safeBalanceOf(address(weth), address(this)) >= amount, "Strategy balance is less than amount");
    TokenUtils.safeApprove(address(weth), address(vault), amount);
    uint256 shares = vault.deposit(amount, address(this));
    uint256 estimatedAmount = vault.convertToAssets(shares);
    // check to ensure minimum amount has been deposited
    // @audit This check uses a withdrawal preview for a deposit, causing potential reverts.
    require(estimatedAmount >= _previewAdjustedWithdraw(amount), "Minimum amount has not been deposited");
    return estimatedAmount;
}
```

This check is problematic for two reasons. First, `_previewAdjustedWithdraw(...)` is semantically intended for estimating withdrawal outputs, not for validating deposit inputs. This is inconsistent with how other strategies perform the slippage check within `_allocate(...)`.

Second, due to the bug in `_previewAdjustedWithdraw(...)`, which can return an inflated value that exceeds the original `amount` if fees are involved, this check can fail. This means that valid allocations can revert, preventing the strategy from depositing funds into the vault. This is specifically possible if the vault has withdrawal fees and `slippageBPS` is set to `0`.

Recommendation(s): Consider refactoring the slippage check to compare the resulting `estimatedAmount` against the original `amount` adjusted by the `slippageBPS`.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.13 [Medium] Unhandled TOKE rewards due to incorrect reward handling during deallocation

File(s): src/TokeAutoUSDStrategy.sol, src/TokeAutoEthStrategy.sol

Description: In both Tokemak strategies, the `_deallocate` function withdraws shares from the Tokemak `Rewarder` contract with the `claim` flag set to `true`. This causes any accrued TOKE rewards to be claimed at the same time as the position is withdrawn.

```
function _deallocate(uint256 amount) internal override returns (uint256) {
    // ...
    // @audit: `claim` is set to true
    rewarder.withdraw(address(this), sharesNeeded, true);

    autoUSD.redeem(sharesNeeded, address(this), address(this));
    // ...
}
```

While the withdrawal of the underlying position succeeds, the simultaneously claimed TOKE rewards are transferred to the strategy contract and remain unprocessed. The deallocation logic does not attempt to swap the received TOKE into the underlying asset, nor does it forward the rewards to the vault. Instead, only the originally requested withdrawal amount is transferred out.

As a result, claimed rewards become locked in the strategy contract and are not reflected in the vault's accounting or realized as yield. Over time, this can lead to accumulating unclaimed value.

Recommendation(s): Consider disabling reward claiming during deallocation by setting the `claim` flag to `false`, and ensure that rewards are claimed exclusively through the dedicated `claimRewards` function.

Status: Unresolved

Update from the client:

6.14 [Medium] Unhandled Tokemak reward staking flow may break reward claiming logic

File(s): src/TokeAutoUSDStrategy.sol, src/TokeAutoEthStrategy.sol

Description: The Tokemak strategies claim accrued rewards by calling the Tokemak `Rewarder.getReward` function inside `_claimRewards`. Depending on the configuration of the `Rewarder` contract, there are two distinct reward distribution flows.

1. **staking rewards is disabled**: In this case, calling `getReward` transfers the accrued rewards directly to the specified recipient in the form of TOKE tokens. Under this configuration, the strategy receives the expected `rewardsClaimed` amount of TOKE, which is then swapped into the underlying asset (USDC or WETH) and forwarded to the vault;
2. **staking rewards is enabled**: (a configuration that can be toggled by Tokemak within the `Rewarder` contract). When staking is enabled, TOKE rewards are not transferred directly to the recipient. Instead, they are automatically staked into Tokemak's staking contract (previously referred to as `accTOKE`, now `sTOKE`). In this flow, the strategy does not receive any TOKE tokens after calling `getReward`;

Because no TOKE is transferred to the strategy, the subsequent swap operation fails due to insufficient token balance. Furthermore, the current strategy implementation does not provide any mechanism to later claim or redeem rewards from the staking contract once the staking period has ended.

```
function _claimRewards(address token, bytes memory quote)
    internal
    override
    returns (uint256 rewardsClaimed)
{
    // ...
    rewardsClaimed = rewarder.earned(address(this));
    // @audit: does not handle staking-enabled reward flow
    rewarder.getReward(address(this), address(MYT), false);

    uint256 amountOut = dexSwap(token, rewardsClaimed, 0, quote);
    TokenUtils.safeTransfer(address(MYT.asset()), address(MYT), amountOut);
}
```

At present, Tokemak has staking rewards disabled, so this issue does not surface. However, since this configuration can be enabled any time in the future by Tokemak, the strategy's reward-claiming logic may break if staking rewards are turned on.

Recommendation(s): Consider updating the logic to handle the case where Tokemak enables staking rewards.

Status: Unresolved

Update from the client:

6.15 [Low] Aave V3 Strategies do not implement rewards claiming from `RewardsController`

File(s): src/strategies/optimism/AaveV3OPUSDCStrategy.sol, src/strategies/arbitrum/AaveV3ARBUSDCStrategy.sol, src/strategies/arbitrum/AaveV3ARBWETHStrategy.sol

Description: The Aave V3 strategies (`AaveV3OPUSDCStrategy`, `AaveV3ARBUSDCStrategy`, and `AaveV3ARBWETHStrategy`) do not implement any mechanism to claim incentive rewards accrued from holding aTokens.

In Aave V3, liquidity suppliers may earn additional incentive tokens through Aave's Rewards Distribution System. These rewards are configured per asset and per network by Aave governance and must be explicitly claimed via the `RewardsController` contract.

As a result, any incentive rewards accrued by these positions will remain unclaimed, leading to a loss of yield for the protocol.

Recommendation(s): Consider implementing the `_claimRewards()` function in all Aave V3 strategies to claim accumulated incentive tokens via Aave's `RewardsController`.

Status: Unresolved

Update from the client:

6.16 [Low] Incorrect unit conversion in `_previewAdjustedWithdraw(...)` leads to inaccurate asset estimates

File(s): WstethMainnetStrategy.sol

Description: The `_previewAdjustedWithdraw(...)` function is intended to estimate how much of the vault's underlying asset (WETH) can be fully withdrawn from the strategy after accounting for slippage, protocol fees, and rounding effects. The function, therefore, expects its input `amount` to be denominated in WETH and should return an adjusted WETH value that reflects the actual deallocation process.

In `WstethMainnetStrategy`, however, the implementation incorrectly treats the input `amount` as if it were denominated in `wstETH` by passing it directly to `getStETHByWstETH(...)`. This does not reflect the real withdrawal flow, which unwraps `wstETH` into `stETH` and then swaps `stETH` for WETH.

Because `wstETH` represents a larger amount of `stETH` over time (due to rebasing), interpreting a WETH-denominated value as `wstETH` results in a systematic overestimation of the amount that can be withdrawn.

```
function _previewAdjustedWithdraw(uint256 amount) internal view override returns (uint256) {
    // @audit The input `amount` is expected to be WETH, but it is treated as wstETH here.
    return wsteth.getStETHByWstETH(amount) - (wsteth.getStETHByWstETH(amount) * slippageBPS / 10_000);
}
```

As a result, the function returns an inflated withdrawal estimate that does not correspond to the actual assets recoverable during deallocation.

Recommendation(s): Consider redesigning the function to accurately simulate the deallocation flow. This involves calculating the equivalent wstETH required, unwrapping it to stETH, and estimating the swap value from stETH to WETH.

Status: Unresolved

Update from the client:

6.17 [Low] Inflated asset estimation in `_previewAdjustedWithdraw`

File(s): src/strategies/*.sol

Description: The `_previewAdjustedWithdraw` function is intended to provide a safe, pessimistic estimate of the assets realizable from a withdrawal. However, the current implementation in multiple strategies inadvertently inflates the base asset value before applying slippage protection. The logic relies on `vault.previewWithdraw(amount)`, which, per ERC-4626 specifications, returns the shares required to receive a net amount of assets. If the vault charges withdrawal fees, the returned share count corresponds to amount + fees.

When these shares are converted back to assets via `vault.convertToAssets(shares)`, the result represents the gross asset value. Consequently, the calculated assets value is strictly greater than or equal to the original amount. This defeats the purpose of `_previewAdjustedWithdraw()`, which should return a conservative estimate to account for slippage and fees.

```
function _previewAdjustedWithdraw(uint256 amount) internal view override returns (uint256) {
    //@audit-issue `previewWithdraw` returns shares worth amount + withdrawal fees
    uint256 shares = vault.previewWithdraw(amount);
    uint256 assets = vault.convertToAssets(shares);
    return assets - (assets * slippageBPS / 10_000);
}
```

NETHERMIND DRAFT

Recommendation(s): Consider updating the calculation to ensure it returns a truly conservative estimate. The implementation should account for fees by deducting them from the realizable amount.

Status: Unresolved

Update from the client:

6.18 [Low] Moonwell strategies do not handle incentive rewards

File(s): `src/strategies/optimism/MoonwellWETHStrategy.sol`, `src/strategies/optimism/MoonwellUSDCStrategy.sol`

Description: The Moonwell strategies (`MoonwellWETHStrategy` and `MoonwellUSDCStrategy`) do not implement any mechanism to handle incentive rewards distributed by the Moonwell protocol.

Moonwell incentivizes suppliers through its `Comptroller` contract, which distributes reward tokens to users who supply assets to the protocol. Rewards can be paid in various tokens, including the underlying asset, WELL, OP tokens, or other configured incentives, depending on the market and current reward programs.

However, Moonwell's reward claiming is not access-controlled; anyone can call `claimReward()` on the `Comptroller` to claim rewards on behalf of any address, including the strategy contract. This means rewards may be sent to the strategy at any time by external parties.

Since the strategies have no mechanism to handle received reward tokens, any rewards claimed (whether by the protocol itself or by external parties on the strategy's behalf) will remain stuck in the contract.

Recommendation(s): Consider implementing a rewards-handling mechanism in both Moonwell strategies. This should include claiming accrued incentives via the Moonwell Comptroller and converting any received reward tokens into the strategy's underlying asset so they can be accounted for by the MYT vault.

Status: Unresolved

Update from the client:

6.19 [Low] Morpho V2 vault `maxRate` and other parameters are not set

File(s): `script/*`

Description: The Morpho V2 vault features a `maxRate` configuration used to ensure the vault's share price does not increase faster than a specified rate. This protects against sudden spikes in the Meta Yield Token (`MYT`) price during external shocks or strategy errors. Additionally, other notable parameters remain unset in the deployment scripts: the `forceDeallocatePenalty` is not configured, which allows users to grief strategies by forcing deallocations at no cost, and the `timelock` values for sensitive functions default to `0`, allowing administrative actions to bypass any safety delay.

Recommendation(s): Consider setting a `maxRate` and `forceDeallocatePenalty`, and configuring the `timelock` mapping for sensitive action types during the deployment process.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.20 [Low] The `WstethMainnetStrategy` does not validate that the `dexSwap(...)` output meets the requested `amount`

File(s): src/strategies/WstethMainnet.sol

Description: The `_deallocate(...)` function in the `WstethMainnetStrategy` contract is responsible for unwrapping `wstETH` into `stETH` and then swapping that `stETH` for `WETH` to satisfy a withdrawal request from the `VaultV2`.

Unlike other strategies that interact with yield-bearing vaults directly (e.g., `EulerWETHStrategy`), this implementation relies on an external swap via `dexSwap(...)`. However, the function ignores the return value of the swap and does not validate that the amount of `WETH` actually received is greater than or equal to the `amount` requested by the vault.

```
function _deallocate(uint256 amount, bytes memory callData, uint256 minIntermediateOut)
    internal
    override
    returns (uint256)
{
    // ...
    wsteth.unwrap(wstETHToUnwrap);
    // ...
    uint256 stETHReceived = stETHAfter - stETHBefore;

    // @audit-issue The return value of dexSwap is ignored
    // and not compared against amount.
    // Swap stETH -> WETH via 0x (will return >= amount due to quote)
    dexSwap(address(weth), address(steth), stETHReceived, 0, callData);

    TokenUtils.safeApprove(address(weth), msg.sender, amount);

    return amount;
}
```

If the swap yields fewer tokens than `amount` (e.g., due to slippage or malicious `callData`), the strategy will still attempt to approve `amount` to the vault. If the strategy happens to have a pre-existing `WETH` balance from other sources, it will inadvertently use those funds to cover the deficit. This creates an inconsistency in fund accounting and, as noted in higher-severity findings, can be made worse when triggered via permissionless functions like `forceDeallocate(...)`, as outlined in *[High] Permissionless `forceDeallocate(...)` combined with missing slippage protection enables sandwich attacks on strategy swaps*.

Recommendation(s): Consider capturing the return value of the `dexSwap(...)` call and adding a requirement to ensure that the received assets are at least equal to the `amount` parameter.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.21 [Low] The `dexSwap(...)` function fails to return the swapped amount, resulting in incorrect reward accounting

File(s): src/MYTStrategy.sol

Description: The `dexSwap(...)` function in the `MYTStrategy` contract is designed to exchange reward tokens for the vault's underlying asset using an external swap provider. While the function signature specifies that it returns a `uint256` value, the implementation lacks a `return` statement. In Solidity, if a function defines a return type but does not explicitly return a value, it defaults to returning the initial value, which is `0` for `uint256`.

```
function dexSwap(...) internal returns (uint256)
{
    // ...
    IERC20(token).approve(allowanceHolder, 0);
    // @audit The function is missing a return statement for the calculated amount.
}
```

This issue directly impacts the reward collection logic in both `TokeAutoUSDStrategy` and `TokeAutoEthStrategy`. Both strategies call `dexSwap(...)` during the `_claimRewards(...)` process to determine how many assets were received and should be transferred back to the main vault.

```
function _claimRewards(address token, bytes memory quote) internal override returns (uint256 rewardsClaimed) {
    // ...
    rewardsClaimed = rewarder.earned(address(this));
    rewarder.getReward(address(this), address(MYT), false);
    // @audit amountOut will always be 0 because dexSwap does not return a value.
    uint256 amountOut = dexSwap(token, rewardsClaimed, 0, quote);
    // @audit Transfers 0 tokens to the vault; rewards remain stuck in the strategy.
    TokenUtils.safeTransfer(address(MYT.asset()), address(MYT), amountOut);
}
```

As a result, `amountOut` is always 0, and no assets are transferred to the vault during this step, even if the swap itself executes successfully.

Recommendation(s): Consider adding an explicit return statement at the end of the `dexSwap(...)` function that returns the difference between `balanceAfter` and `balanceBefore`.

Status: Unresolved

Update from the client:

6.22 [Info] Absence of slippage check in `dexSwap` due to unused `minAmountOut` parameter

File(s): src/MYTStrategy.sol

Description: The `dexSwap()` function in the `MYTStrategy` contract accepts a `minAmountOut` parameter intended to provide slippage protection for token swaps. However, this parameter is never actually used to validate the amount received from the swap operation.

```
function dexSwap(address to, address from, uint256 amount, uint256 minAmountOut, bytes memory callData) internal
↪  returns (uint256) {
    IERC20(from).approve(allowanceHolder, type(uint256).max);
    uint256 targetBalanceBefore = IERC20(to).balanceOf(address(this));
    (bool success, ) = allowanceHolder.call(callData);
    require(success, "0x exception");
    uint256 targetBalanceAfter = IERC20(to).balanceOf(address(this));
    IERC20(from).approve(allowanceHolder, 0);
    return targetBalanceAfter - targetBalanceBefore;
}
```

The function calculates the amount received (`targetBalanceAfter - targetBalanceBefore`) but never validates it against the `minAmountOut` parameter. This means that regardless of what `minAmountOut` value is passed, the swap will execute with zero slippage protection.

Recommendation(s): Consider adding a validation to ensure the actual amount received meets the minimum expected amount `minAmountOut`.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.23 [Info] Deployment script for Fluid strategy uses wrong address

File(s): `script/DeployV3Arb.s.sol`

Description: The deployment script for the Arbitrum strategies specifies that the Fluid strategy should be deployed with a vault address of `0xeAbBfca72F8a8bf14C4ac59e69ECB2eB69F0811C`, which points to a deprecated vault which doesn't support ERC4626, and as a result, the strategy will not operate correctly. For a Fluid USDC strategy on Arbitrum that supports ERC4626, it seems that the correct `fToken` address to use would be `0x1A996cb54bb95462040408C06122D45D6Cdb6096`.

Recommendation(s): Consider adjusting the deployment script to use a Fluid ERC4626 compatible vault.

Status: Unresolved

Update from the client:

6.24 [Info] Strategies may not account for entire native asset amount

File(s): `src/strategies/WstethMainnet.sol`, `src/strategies/optimism/MoonwellWETHStrategy.sol`

Description: Both `WstethMainnetStrategy` and `MoonwellWETHStrategy` implement a `receive()` function that accepts ETH from any sender. These strategies are only expected to receive ETH from the WETH contract during specific internal operations:

- In `WstethMainnetStrategy._allocate()`, WETH is unwrapped into ETH prior to depositing into Lido;
- In `MoonwellWETHStrategy._deallocate()`, ETH is received and immediately wrapped back into WETH;

Outside of these controlled flows, the contract can still accept ETH, which can lead to a scenario where the strategy has more ETH than expected. If this happens, the additional ETH is ignored, and the native assets are wrapped/unwrapped depending on the function arguments. Consider this code snipped from the `WstethMainnet._allocate(...)` function:

```
function _allocate(uint256 amount) ... returns (uint256 depositReturn) {
    // ...
    // Unwrap WETH to ETH
    weth.withdraw(amount);
    // @audit We only deposit `amount` ETH into Lido's stETH contract
    //        Any additional ETH beyond the `amount` is not used
    uint256 stETHReceived = steth.submit{value: amount}(address(0));
    // ...
}
```

Recommendation(s): When converting the native assets back into a wrapped asset or when depositing native assets, consider using `address(this).balance` to provide the entire native asset amount rather than relying on the function arguments.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.25 [Info] The `proxy(...)` function allows operators to bypass access controls in `AlchemistCurator`

File(s): src/utils/PermissionedProxy.sol, src/AlchemistCurator.sol

Description: The `PermissionedProxy` contract serves as a base for access control, defining `admin` and `operator` roles. It includes a `proxy(...)` function designed to allow operators to execute arbitrary calls on the vault, subject to restrictions defined in the `permissionedCalls` mapping.

The `AlchemistCurator` contract inherits from `PermissionedProxy` and contains sensitive configuration functions restricted to the `onlyAdmin` modifier. These include functions for managing the vault, such as `increaseAbsoluteCap(...)`, `submitSetAllocator(...)`, and strategy management functions.

However, the validation logic within the `proxy(...)` function operates as a blacklist rather than a whitelist. It checks if a selector is *not* in the mapping, essentially allowing any function call by default unless it is explicitly restricted.

```
function proxy(address vault, bytes memory data) external payable onlyOperator {
    bytes4 selector;
    require(data.length >= 4, "SEL");
    assembly {
        selector := mload(add(data, 32))
    }
    // @audit The logic acts as a blacklist.
    // If the selector is not in the mapping, it passes.
    require(!permissionedCalls[selector], "PD");

    (bool success, ) = vault.call{value: msg.value}(data);
    require(success, "failed");
}
```

While the `AlchemistAllocator` contract explicitly populates the `permissionedCalls` mapping to prevent operators from calling `allocate` and `deallocate` directly via the proxy, the `AlchemistCurator` contract does not initialize this mapping with any selectors.

This configuration allows an operator to bypass the `onlyAdmin` restrictions within the `AlchemistCurator`. By encoding a call to a restricted vault function (e.g., `increaseAbsoluteCap`) and passing it to the `proxy(...)` function, the operator can execute the transaction. Since the `AlchemistCurator` contract itself is the authorized entity on the vault, the vault accepts the call.

This effectively grants operators admin-level privileges over the vault configuration managed by the curator. Furthermore, the blacklist approach is fragile, as any new function added to the system is accessible to operators by default unless explicitly restricted.

Recommendation(s): Consider populating the `permissionedCalls` mapping in the `AlchemistCurator` constructor to blacklist sensitive function selectors, ensuring operators cannot bypass access controls. Alternatively, consider refactoring the `permissionedCalls` logic to function as a whitelist, granting operators access only to explicitly approved functions.

Status: Unresolved

Update from the client:

6.26 [Info] TokeAuto strategies optimistically preview withdrawals

File(s): src/TokeAutoUSDStrategy.sol, src/TokeAutoEthStrategy.sol

Description: Both TokeAuto strategies implement the function `_previewAdjustedWithdraw`, which is used to preview the withdrawal amount with some margin for slippage. Both of these functions are implemented as follows:

```
function _previewAdjustedWithdraw(uint256 amount) ... returns (uint256) {
    uint256 sharesNeeded = auto.convertToShares(amount);
    uint256 assets = auto.convertToAssets(sharesNeeded);
    return assets - (assets * slippageBPS / 10_000);
}
```

The conversion from assets to shares and back uses the regular ERC4626 vault functions; however, Autopools feature additional logic to calculate withdrawals or deposits differently using an additional `TotalAssetsPurpose` argument. When a regular `convertToShares` or `convertToAssets` call is executed, the Autopool will default to using `TotalAssetsPurpose.Global`. This means the `_previewAdjustedWithdrawal` function is calculating the asset withdrawal amount using the `Global` flag instead of the `Withdraw` flag, which would be used during a withdrawal. When this function is called, it will return an optimistic withdrawal preview that is higher than the actual expected amount.

Recommendation(s): Consider changing the `convertToShares` and `convertToAssets` calls to also provide a `TotalAssetsPurpose` of `Withdraw` to make `_previewAdjustedWithdraw` return an accurate withdrawal amount.

Status: Unresolved

Update from the client:

6.27 [Info] Unnecessary timelock pattern for cap decrease functions in `AlchemistCurator`

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

Description: The MYT vault implements a timelock mechanism for sensitive curator operations. It requires the curator to first submit the action through `submit()`, wait for the timelock period, and then execute it. However, the vault intentionally does not enforce a timelock for `decreaseAbsoluteCap` and `decreaseRelativeCap`; these functions can be called immediately by the curator or sentinel without going through the submit-execute flow.

Despite this, the `AlchemistCurator` contract exposes both direct execution and submit-execute patterns for these decrease functions. This can be misleading: an operator using `submitDecreaseAbsoluteCap` or `submitDecreaseRelativeCap` would expect a timelock period before execution, but the vault will accept immediate execution anyway. The submit functions add unnecessary complexity and may cause operators to wait for a nonexistent timelock.

Recommendation(s): Consider removing the `submitDecreaseAbsoluteCap` and `submitDecreaseRelativeCap` functions from the `AlchemistCurator` contract to avoid confusion.

Status: Unresolved

Update from the client:

6.28 [Info] Unsupported action types silently return 0 instead of reverting

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

Description: The `MYTStrategy` base contract defines multiple overloaded versions of `_allocate` and `_deallocate` to support different action types (`direct`, `swap`, `unwrapAndSwap`):

```
function _allocate(uint256 amount) internal virtual returns (uint256) {}
function _allocate(uint256 amount, bytes memory callData) internal virtual returns (uint256) {}

function _deallocate(uint256 amount) internal virtual returns (uint256) {}
function _deallocate(uint256 amount, bytes memory callData) internal virtual returns (uint256) {}
function _deallocate(
    uint256 amount,
    bytes memory callData,
    uint256 minIntermediateOutAmount
) internal virtual returns (uint256) {}
```

All of these functions have empty implementations in the base contract and therefore implicitly return `0`. Concrete strategy implementations override only the variants corresponding to the action types they support.

The public `allocate()` and `deallocate()` entry points dispatch to these internal functions based on the `ActionType` provided by the caller.

If a caller specifies an action type that the strategy does not support, the call falls back to the empty base implementation rather than reverting. As a result, the function appears to succeed but performs no allocation or deallocation and returns `0`.

Recommendation(s): Consider modifying the base implementations of these virtual functions to revert by default. Alternatively, consider making the `MYTStrategy` base contract `abstract` and leaving the functions unimplemented, which would force all inheriting strategies to explicitly define the behavior for each variant.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.29 [Info] Unused `minFinalOut` parameter in `MYTStrategy`

File(s): src/MYTStrategy.sol

Description: The `SwapParams` struct includes a `minFinalOut` parameter, designed to provide slippage protection by enforcing a minimum output amount during swap-based deallocations. While the `AlchemistAllocator` correctly populates this value when calling `deallocateWithSwap` or `deallocateWithUnwrapAndSwap`, the `MYTStrategy` contract fails to pass it to the strategy implementations:

```
function deallocate(...)
    external
    onlyVault
    returns (bytes32[] memory strategyIds, int256 change)
{
    // ...
    if (action == ActionType.swap) {
        amountDeallocated = _deallocate(assets, adapterParams.swapParams.txData);
    } else if (action == ActionType.unwrapAndSwap) {
        amountDeallocated = _deallocate(assets, adapterParams.swapParams.txData,
        ↪  adapterParams.swapParams.minIntermediateOut);
    }
    // ...
}
```

Recommendation(s): Ensure that `minFinalOut` is passed to the strategy's `_deallocate` implementations to enforce the intended slippage limits. Alternatively, remove it from the `SwapParams` struct if it's not intended to be used.

Status: Unresolved

Update from the client:

6.30 [Info] `AlchemistCurator` doesn't support all curator functions

File(s): src/AlchemistCurator.sol

Description: The `AlchemistCurator` contract is used to interact with curator-only functions on the MYT vault. The following is a list of functions that a MYT vault curator should be able to access directly:

```
revoke                          <-- Not supported by AlchemistCurator
decreaseAbsoluteCap
decreaseRelativeCap
```

The `AlchemistCurator` can also access some functions indirectly through the `submit` function, which allows the intended function to be called after a timelock. These functions are shown below:

```
setReceiveSharesGate            <-- Not supported by AlchemistCurator
setSendSharesGate               <-- Not supported by AlchemistCurator
setReceiveAssetsGate            <-- Not supported by AlchemistCurator
setSendAssetsGate               <-- Not supported by AlchemistCurator
setAdapterRegistry              <-- Not supported by AlchemistCurator
addAdapter
removeAdapter
increaseTimelock                <-- Not supported by AlchemistCurator
decreaseTimelock                <-- Not supported by AlchemistCurator
abdicate                        <-- Not supported by AlchemistCurator
setPerformanceFee               <-- Not supported by AlchemistCurator
setManagementFee                <-- Not supported by AlchemistCurator
setPerformanceFeeRecipient      <-- Not supported by AlchemistCurator
setManagementFeeRecipient       <-- Not supported by AlchemistCurator
increaseAbsoluteCap
decreaseAbsoluteCap
increaseRelativeCap
decreaseRelativeCap
setForceDeallocatePenalty       <-- Not supported by AlchemistCurator
```

Recommendation(s): Consider explicitly implementing the necessary curator functions in `AlchemistCurator`.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

6.31 [Info] `slippageBPS` cannot be updated after deployment

File(s): src/MYTStrategy.sol

Description: The `MYTStrategy` uses the `slippageBPS` value to determine the acceptable slippage when withdrawing or allocating in certain strategies. Eventually, in `TokeAutoEthStrategy` and `TokeAutoUSDStrategy`, this value is used during allocation to validate that received assets meet the minimum threshold.

However, during `MYTStrategy` initialization, the `slippageBPS` is stored twice in the contract: once as part of the `params` struct and again as a standalone state variable. Additionally, the contract does not provide any function to update this value after deployment. As a result, if market conditions change and a different slippage tolerance is required, the only way to update this parameter is to redeploy the entire strategy.

```
constructor(address _myt, StrategyParams memory _params) Ownable(_params.owner) {
    // ...
    require(_params.slippageBPS < 1000);
    MYT = IVaultV2(_myt);

    params = _params;
    adapterId = keccak256(abi.encode("this", address(this)));
    // @audit duplicated storage (also stored in params)
    slippageBPS = _params.slippageBPS;

    allowanceHolder = 0x0000000000001fF3684f28c67538d4D072C22734;
}
```

Recommendation(s): Consider eliminating the duplicate storage of `slippageBPS` and introducing a function to update the slippage tolerance after deployment.

Status: Unresolved

Update from the client:

6.32 [Best Practices] Unnecessary interface declarations

File(s): src/*

Description: The codebase features some interface declarations that are unused, or could have been imported from the openzeppelin-contracts instead. The following is a list of such interfaces:

```
    interface IERC721Tiny (unused, even if used, could import)
    import ISignatureTransfer (unused)
    import forge-std/console.sol (unused)

src/external/interfaces/IDetailedERC20.sol
    interface IDetailedERC20 (unused, even if used, could import)

src/external/interfaces/ISettlerActions.sol
    interface ISettlerActions (unused)

src/external/interfaces/IVelodromePair.sol
    interface IVelodromePair (unused, even if used, could import)

src/strategies/mainnet/MorphoYearnOGWETH.sol
    interface IERC4626 (import from OZ instead)
    import IMYTStrategy (unused)
```

Recommendation(s): Consider removing unused imports and using existing dependency interfaces for the points listed above.

Status: Unresolved

Update from the client:

NETHERMIND DRAFT

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 Alchemix's documentation

The documentation for the `Alchemix` contracts was provided through the code's Natspec and inline comments. Moreover, the `Alchemix` team provided a system overview explanation during the kickoff call and has addressed all questions and concerns raised by the Nethermind Security team during their regular calls, providing valuable insights and a comprehensive understanding of the project's technical aspects.

NETHERMIND DRAFT

8 Test Suite Evaluation

Remarks about the Alchemix V3 Test Suite

The current test suite provides a basic framework for strategy unit tests but lacks the depth required for a protocol of this complexity. Most tests are repetitive and focus on simple success cases rather than verifying the integration logic needed for safe fund management.

A significant portion of the identified vulnerabilities, including those preventing core strategy functionality, could have been identified through basic end-to-end testing. At the start of the review, interactions between the `AlchemistAllocator` and target strategies were notably impaired. This gap likely stems from codebase fragmentation due to licensing requirements, which hindered a unified testing environment. Consequently, the test suite focused primarily on the happy path, lacking sufficient coverage of real-world integration logic and actual deployment configurations.

Moving forward, it is strongly recommended to prioritize high-level end to end tests for every strategy. These tests should simulate multiple cycles of allocation and deallocation with significant time delays to ensure interest accrual and yield snapshots behave correctly. Furthermore, the reward claiming logic, specifically the conversion of incentives back to the underlying asset, requires rigorous testing to ensure funds do not become trapped in the adapters.

To bridge the gap between development and production, all end to end tests should be executed against the exact deployment scripts intended for mainnet use. Finally, a public testnet deployment should be considered. Stress-testing the protocol under real world conditions and the unexpected chaos of live user interaction is a vital step before committing significant capital to the mainnet.

8.1 Tests Output

```
Ran 10 tests for test/AlchemistStrategyClassifier.t.sol:AlchemistStrategyClassifierTest
[PASS] testAcceptOwnership() (gas: 34208)
[PASS] testAcceptOwnershipNoPendingAdminRevert() (gas: 14566)
[PASS] testAcceptOwnershipUnauthorizedRevert() (gas: 42831)
[PASS] testAssignStrategyRiskLevel() (gas: 37985)
[PASS] testAssignStrategyRiskLevelMultipleStrategies() (gas: 41124)
[PASS] testAssignStrategyRiskLevelUnauthorizedRevert() (gas: 14750)
[PASS] testSetRiskClass() (gas: 30583)
[PASS] testSetRiskClassMultipleClasses() (gas: 40421)
[PASS] testTransferOwnership() (gas: 40126)
[PASS] testTransferOwnershipUnauthorizedRevert() (gas: 16783)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 3.68ms (2.78ms CPU time)

Ran 7 tests for test/strategies/WstethMainnetStrategy.t.sol:WstethMainnetStrategyTest
[PASS] test_allocator_allocate(uint256) (runs: 256, : 316125, ~: 316127)
[FAIL: wstETH balance should be positive: 0 <= 0] test_allocator_allocate_with_swap() (gas: 492752)
[FAIL: wstETH balance should be positive: 0 <= 0] test_allocator_deallocate_with_swap() (gas: 484042)
[PASS] test_strategy_allocate_direct() (gas: 349913)
[FAIL: change is less than amount: 0 <= 815068212560160000000] test_strategy_allocate_with_swap() (gas: 604644)
[FAIL: Zero amount] test_strategy_deallocate_with_swap() (gas: 636470)
[FAIL: Zero amount] test_vault_deallocate_from_strategy_with_bidirectional_swap() (gas: 513902)
Suite result: FAILED. 2 passed; 5 failed; 0 skipped; finished in 2.89s (378.03ms CPU time)

Ran 11 tests for test/strategies/PeapodsUSDCStrategy.t.sol:PeapodsUSDCStrategyTest
[PASS] testConfig() (gas: 11057)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 227958, ~: 228026)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 289735, ~: 289753)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213631)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189615)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 340246, ~: 340270)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 272377)
[PASS] test_strategy_full_deallocate(uint256) (runs: 256, : 332866, ~: 332892)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 216148, ~: 216179)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 195642, ~: 195729)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 262441, ~: 262461)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 4.38s (1.96s CPU time)

Ran 11 tests for test/strategies/AaveV3OPUSDCStrategy.t.sol:AaveV3OPUSDCStrategyTest
[PASS] testConfig() (gas: 11101)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 383245, ~: 383316)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 474335, ~: 474356)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213342)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189300)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 492346, ~: 492361)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 407509, ~: 407525)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 392095)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 215473, ~: 215486)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 349162, ~: 349237)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 438480, ~: 438502)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 6.16s (3.73s CPU time)
```

NETHERMIND DRAFT

```
Ran 9 tests for test/AlchemistAllocator.t.sol:AlchemistAllocatorTest
[PASS] testAllocate() (gas: 488166)
[PASS] testAllocateRevertIfAboveAbsoluteCap() (gas: 328849)
[PASS] testAllocateRevertIfAboveRelativeCap() (gas: 335095)
[PASS] testAllocateRevertIfAboveRiskGlobalCap() (gas: 336047)
[PASS] testAllocateRevertIfAboveRiskIndividualCap() (gas: 339239)
[PASS] testAllocateUnauthorizedAccessRevert() (gas: 13763)
[PASS] testDeallocate() (gas: 558461)
[PASS] testDeallocateUnauthorizedAccessRevert() (gas: 13763)
[PASS] testDeallocateWithYield() (gas: 797526)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 16.95ms (14.84ms CPU time)

Ran 31 tests for test/AlchemistCurator.t.sol:AlchemistCuratorTest
[PASS] testAcceptAdminOwnershipUnauthorizedAccessRevert() (gas: 11532)
[PASS] testDecreaseAbsoluteCap() (gas: 277765)
[PASS] testDecreaseAbsoluteCapReverOnInvalidAdapter() (gas: 25853)
[PASS] testDecreaseAbsoluteCapUnauthorizedAccessRevert() (gas: 15876)
[PASS] testDecreaseRelativeCap() (gas: 275028)
[PASS] testDecreaseRelativeCapReverOnInvalidAdapter() (gas: 25479)
[PASS] testDecreaseRelativeCapUnauthorizedAccessRevert() (gas: 16162)
[PASS] testIncreaseAbsoluteCap() (gas: 221027)
[PASS] testIncreaseAbsoluteCapReverOnInvalidAdapter() (gas: 26502)
[PASS] testIncreaseAbsoluteCapUnauthorizedAccessRevert() (gas: 16070)
[PASS] testIncreaseRelativeCap() (gas: 219250)
[PASS] testIncreaseRelativeCapReverOnInvalidAdapter() (gas: 25919)
[PASS] testIncreaseRelativeCapUnauthorizedAccessRevert() (gas: 16697)
[PASS] testSetStrategy() (gas: 141740)
[PASS] testSetStrategyInvalidAdapterRevert() (gas: 16714)
[PASS] testSetStrategyInvalidMYTRevert() (gas: 46837)
[PASS] testSetStrategyUnauthorizedAccessRevert() (gas: 15896)
[PASS] testSubmitDecreaseAbsoluteCap() (gas: 189648)
[PASS] testSubmitDecreaseAbsoluteCapReverOnInvalidAdapter() (gas: 25752)
[PASS] testSubmitDecreaseAbsoluteCapRevertUnauthorizedAccess() (gas: 16297)
[PASS] testSubmitDecreaseRelativeCap() (gas: 190176)
[PASS] testSubmitDecreaseRelativeCapReverOnInvalidAdapter() (gas: 26720)
[PASS] testSubmitDecreaseRelativeCapRevertUnauthorizedAccess() (gas: 16891)
[PASS] testSubmitIncreaseAbsoluteCap() (gas: 189472)
[PASS] testSubmitIncreaseAbsoluteCapReverOnInvalidAdapter() (gas: 26236)
[PASS] testSubmitIncreaseAbsoluteCapUnauthorizedAccessRevert() (gas: 15809)
[PASS] testSubmitIncreaseRelativeCap() (gas: 190132)
[PASS] testSubmitIncreaseRelativeCapReverOnInvalidAdapter() (gas: 25840)
[PASS] testSubmitIncreaseRelativeCapUnauthorizedAccessRevert() (gas: 15985)
[PASS] testSubmitSetStrategy() (gas: 55414)
[PASS] testTransferAdminOwnerShipUnauthorizedAccessRevert() (gas: 12383)
Suite result: ok. 31 passed; 0 failed; 0 skipped; finished in 21.11ms (19.87ms CPU time)

Ran 11 tests for test/strategies/AaveV3ARBUSDCStrategy.t.sol:AaveV3ARBUSDCStrategyTest
[PASS] testConfig() (gas: 11101)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 388385, ~: 388458)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 479866, ~: 479875)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213324)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189282)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 497932, ~: 497946)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 412033, ~: 412048)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 396279)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 215455, ~: 215466)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 354390, ~: 354423)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 444034, ~: 444065)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 6.28s (3.86s CPU time)

Ran 11 tests for test/strategies/StargateEthPoolStrategy.t.sol:StargateEthPoolStrategyTest
[PASS] testConfig() (gas: 11057)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 216238, ~: 216240)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 319598, ~: 319806)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 204847)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 180787)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 344279, ~: 344303)
[PASS] test_strategy_deallocate_full_amount(uint256) (runs: 256, : 335769, ~: 335788)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 251108)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 200320, ~: 200344)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 186055, ~: 186163)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 287652, ~: 287932)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 2.20s (1.90s CPU time)

Ran 6 tests for test/MYTStrategy.t.sol:MYTStrategyTest
[PASS] test_allocatorCanAllocateAndDeallocate() (gas: 33833)
[PASS] test_onlyWhitelistedAllocatorCanAllocate() (gas: 29849)
[PASS] test_onlyWhitelistedAllocatorCanDeallocate() (gas: 28490)
[PASS] test_strategyIntegrationWithAlchemist() (gas: 383054)
[PASS] test_strategyParametersCanBeUpdated() (gas: 80569)
[PASS] test_strategyRespectsAlchemistPauseStates() (gas: 321286)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 326.43ms (1.45ms CPU time)
```

NETHERMIND DRAFT

```
Ran 11 tests for test/strategies/TokeAutoUSDStrategy.t.sol:TokeAutoUSDStrategyTest
[PASS] testConfig() (gas: 11079)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 438030, ~: 438052)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 587716, ~: 587727)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 215183)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 191174)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 617638, ~: 617652)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 459778)
[PASS] test_strategy_full_deallocate(uint256) (runs: 256, : 606359, ~: 606374)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 217736, ~: 217750)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 398901, ~: 398985)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 549099, ~: 549102)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 8.41s (5.90s CPU time)

Ran 13 tests for test/strategies/MoonwellWETHStrategy.t.sol:MoonwellWETHStrategyTest
[PASS] testConfig() (gas: 11057)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 440992, ~: 441053)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 603639, ~: 603644)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 204792)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 180768)
[PASS] test_strategy_allocate_reverts_on_mint_failure() (gas: 206653)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 617825, ~: 617867)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 435577)
[PASS] test_strategy_deallocate_reverts_on_redeem_failure() (gas: 438396)
[PASS] test_strategy_full_deallocate(uint256) (runs: 256, : 575736, ~: 575744)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 200262, ~: 200301)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 405451, ~: 405534)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 566300, ~: 566307)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 8.60s (6.18s CPU time)

Ran 11 tests for test/strategies/FluidARBUSDCStrategy.t.sol:FluidARBUSDCStrategyTest
[PASS] testConfig() (gas: 11101)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 273039, ~: 273124)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 353264, ~: 353285)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213275)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189294)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 397967, ~: 397977)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 317071, ~: 317084)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 309272)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 215442, ~: 215443)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 239378, ~: 239411)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 322568, ~: 322577)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 11.22s (5.50s CPU time)

Ran 11 tests for test/strategies/EulerWETHStrategy.t.sol:EulerWETHStrategyTest
[PASS] testConfig() (gas: 11079)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 310254, ~: 310372)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 418866, ~: 418905)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 205088)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 181089)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 447757, ~: 447766)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 353471, ~: 353511)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 330093)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 201088, ~: 201089)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 272445, ~: 272530)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 379267, ~: 379267)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 3.62s (3.68s CPU time)

Ran 11 tests for test/strategies/EulerARBWETHStrategy.t.sol:EulerARBWETHStrategyTest
[PASS] testConfig() (gas: 11079)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 332797, ~: 332839)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 449487, ~: 449494)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213459)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189460)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 467294, ~: 467312)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 368709, ~: 368754)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 345331)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 213259, ~: 213302)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 292808, ~: 292834)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 407745, ~: 407736)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 6.17s (4.54s CPU time)

Ran 11 tests for test/strategies/EulerUSDCStrategy.t.sol:EulerUSDCStrategyTest
[PASS] testConfig() (gas: 11079)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 328318, ~: 328326)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 443608, ~: 443619)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213631)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189632)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 464920, ~: 464933)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 367164, ~: 367188)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 343764)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 216194, ~: 216200)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 289499, ~: 289540)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 403054, ~: 403058)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 4.15s (4.30s CPU time)
```

NETHERMIND DRAFT

```
Ran 11 tests for test/strategies/EulerARBUSDCStrategy.t.sol:EulerARBUSDCStrategyTest
[PASS] testConfig() (gas: 11079)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 340118, ~: 340215)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 459965, ~: 459976)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213275)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189276)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 472168, ~: 472182)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 372578, ~: 372586)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 347338)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 215395, ~: 215411)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 299031, ~: 299074)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 417046, ~: 417060)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 14.84s (5.37s CPU time)

Ran 11 tests for test/strategies/PeapodsETHStrategy.t.sol:PeapodsETHStrategyTest
[PASS] testConfig() (gas: 11090)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 840062, ~: 840102)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 1140822, ~: 1140867)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 205114)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 181098)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 1166992, ~: 1167000)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 838839)
[PASS] test_strategy_full_deallocate(uint256) (runs: 256, : 994527, ~: 994531)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 201107, ~: 201149)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 808531, ~: 808631)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 1107517, ~: 1107535)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 15.13s (20.10s CPU time)

Ran 11 tests for test/strategies/MorphoYearnOGWETHStrategy.t.sol:MorphoYearnOGWETHStrategyTest
[PASS] testConfig() (gas: 11079)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 1969397, ~: 1969516)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 2852910, ~: 2852954)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 205088)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 181089)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 2378135, ~: 2378184)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 1675160, ~: 1675181)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 1467179)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 201109, ~: 201156)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 1770863, ~: 1770956)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 2652647, ~: 2652684)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 11.16s (30.30s CPU time)

Ran 12 tests for test/strategies/TokeAutoETHStrategy.t.sol:TokeAutoETHStrategyTest
[PASS] testConfig() (gas: 11123)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 413293, ~: 413374)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 550479, ~: 550505)
[PASS] test_deallocate_full_real_assets() (gas: 588543)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 205139)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 181130)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 590315, ~: 590320)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 541652, ~: 541663)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 442016)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 201154, ~: 201166)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 376360, ~: 376421)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 519838, ~: 519572)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 15.13s (5.40s CPU time)

Ran 11 tests for test/strategies/AaveV3ARBWETHStrategy.t.sol:AaveV3ARBWETHStrategyTest
[PASS] testConfig() (gas: 11101)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 352718, ~: 352756)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 439435, ~: 439481)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213508)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189466)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 458458, ~: 458501)
[PASS] test_strategy_deallocate_reverts_due_to_slippage(uint256,uint256) (runs: 256, : 382952, ~: 382957)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 368856)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 213303, ~: 213345)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 318669, ~: 318731)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 403650, ~: 403703)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 15.13s (7.35s CPU time)

Ran 13 tests for test/strategies/MoonwellUSDCStrategy.t.sol:MoonwellUSDCStrategyTest
[PASS] testConfig() (gas: 11057)
[PASS] test_allocator_allocate_direct(uint256) (runs: 256, : 467119, ~: 467147)
[PASS] test_allocator_deallocate_direct(uint256,uint256) (runs: 256, : 611379, ~: 611388)
[PASS] test_strategy_allocate_reverts_due_to_paused_allocation() (gas: 213275)
[PASS] test_strategy_allocate_reverts_due_to_zero_amount() (gas: 189259)
[PASS] test_strategy_allocate_reverts_on_mint_failure() (gas: 218150)
[PASS] test_strategy_deallocate(uint256,uint256) (runs: 256, : 622516, ~: 622526)
[PASS] test_strategy_deallocate_reverts_due_to_zero_amount() (gas: 456034)
[PASS] test_strategy_deallocate_reverts_on_redeem_failure() (gas: 460563)
[PASS] test_strategy_full_deallocate(uint256) (runs: 256, : 561071, ~: 561086)
[PASS] test_strategy_withdrawToVault(uint256) (runs: 256, : 215391, ~: 215403)
[PASS] test_vault_allocate_to_strategy(uint256) (runs: 256, : 430817, ~: 430852)
[PASS] test_vault_deallocate_from_strategy(uint256,uint256) (runs: 256, : 573288, ~: 573296)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 15.13s (7.22s CPU time)

Ran 21 test suites in 15.14s (150.96s CPU time): 239 tests passed, 5 failed, 0 skipped (244 total tests)
```

NETHERMIND DRAFT

```
Failing tests:
Encountered 5 failing tests in test/strategies/WstethMainnetStrategy.t.sol:WstethMainnetStrategyTest
[FAIL: wstETH balance should be positive: 0 <= 0] test_allocator_allocate_with_swap() (gas: 492752)
[FAIL: wstETH balance should be positive: 0 <= 0] test_allocator_deallocate_with_swap() (gas: 484042)
[FAIL: change is less than amount: 0 <= 815068212560160000000] test_strategy_allocate_with_swap() (gas: 604644)
[FAIL: Zero amount] test_strategy_deallocate_with_swap() (gas: 636470)
[FAIL: Zero amount] test_vault_deallocate_from_strategy_with_bidirectional_swap() (gas: 513902)

Encountered a total of 5 failing tests, 239 tests succeeded
```

8.2 Automated Tools

8.2.1 AuditAgent

The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

NETHERMIND DRAFT

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.

NETHERMIND DRAFT

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.

NETHERMIND DRAFT