
Security Review Report

NM-0764 - SUNO



NETHERMIND
SECURITY

(March 24, 2026)

Contents

1	Executive Summary	2
2	Audited Files	3
3	Summary of Issues	3
4	System Overview	4
4.1	Staking Core	4
4.2	Reward Distribution Modules	4
4.3	Reward Accrual and Claiming	4
4.4	Auto-Compounding Vault & Strategy	4
4.5	The Harvest Mechanism	5
5	Risk Rating Methodology	6
6	Issues	7
6.1	[Medium] Blocklist bypass via increaseAllowance and transferFrom	7
6.2	[Low] Sequential fee application in harvest(...) results in lower than expected harvester bounties	8
6.3	[Info] Strategy replacement can lead to loss of funds	9
6.4	[Info] The harvest(...) function logic may be inconsistent depending on the Quoter implementation	10
6.5	[Info] The reportRewards function is incompatible with tokens that revert on zero approvals	11
6.6	[Info] Unused code	11
6.7	[Best Practices] Redundant calls to the quoter contract in harvest(...) increase gas consumption	12
7	Documentation Evaluation	13
8	Test Suite Evaluation	14
8.1	Automated Tools	14
8.1.1	AuditAgent	14
9	About Nethermind	15

1 Executive Summary

This document presents the results of the security review conducted by [Nethermind Security](#) for [Suno Protocol's](#) smart contracts.

The Suno Protocol's team has developed a modular staking and auto-compounding protocol centered around the **uWatt** token. The architecture utilizes an ERC4626-compliant vault (UWattsVault4626) that delegates asset management to a specialized strategy contract (StakingStrategy). This strategy optimizes yield by staking assets in the StakingUWatts contract and auto-compounding rewards. A notable feature of the system is its price-routing logic, which compares real-time quotes between a decentralized exchange (Uniswap V3) and a dedicated UWattVendor to ensure the most efficient acquisition of tokens during the harvest process.

Furthermore, the protocol implements a flexible reward distribution mechanism through the IRewarder interface, supporting both traditional index-based accrual via the Claim contract and real-time streaming via the SuperfluidAdapter. This adapter integrates with Superfluid's General Distribution Agreement (GDA) to provide continuous reward flows to stakers. The entire suite is built with upgradeability in mind using the UUPS proxy pattern, allowing for future logic enhancements while maintaining a robust access control system across staking, distribution, and vendor components.

The audit comprises 641 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 7 points of attention, where one is classified Medium, one is classified as Low and five are classified as Informational and 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 summary of issues. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

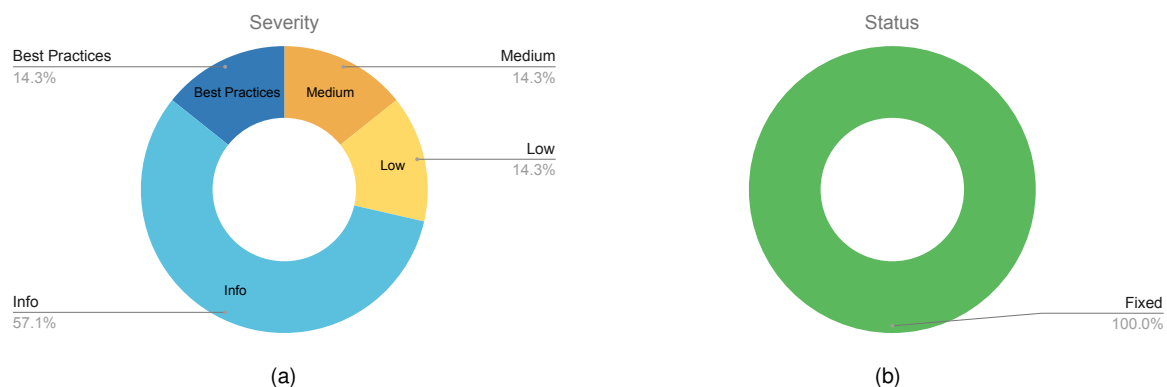


Fig. 1: Distribution of issues: Critical (0), High (0), Medium (1), Low (1), Undetermined (0), Informational (4), Best Practices (1).
Distribution of status: Fixed (7), Acknowledged (0), Mitigated (0), Unresolved (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	February 24, 2026
Final Report	March 24, 2026
Initial Commit	5afd58eafcd7887c6cb9a45712630e53434234b
Final Commit	ee8947005faa13f2339d9d9dbb014d9dd766be5b5
Documentation Assessment	Low
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/staking_module/UWattsVault4626.sol	161	117	72.7%	37	315
2	contracts/staking_module/StakingUWatts.sol	216	148	68.5%	36	400
3	contracts/staking_module/StakingStrategy.sol	264	183	69.3%	50	497
	Total	641	448	69.9%	123	1212

3 Summary of Issues

	Finding	Severity	Update
1	Blocklist bypass via increaseAllowance and transferFrom	Medium	Fixed
2	Sequential fee application in harvest(...) results in lower than expected harvester bounties	Low	Fixed
3	Strategy replacement can lead to loss of funds	Info	Fixed
4	The harvest(...) function logic may be inconsistent depending on the Quoter implementation	Info	Fixed
5	The reportRewards function is incompatible with tokens that revert on zero approvals	Info	Fixed
6	Unused code	Info	Fixed
7	Redundant calls to the quoter contract in harvest(...) increase gas consumption	Best Practices	Fixed

4 System Overview

The Suno Staking protocol is a yield-generation and auto-compounding system designed for uWatt token holders. The architecture is built around a core staking contract and extends into an ERC4626-compliant vault that automates reward reinvestment. To provide flexibility in yield distribution, the protocol employs a modular reward system, allowing administrators to switch between index-based claiming and continuous real-time token streaming via the **Superfluid** protocol.

4.1 Staking Core

The StakingUWatts contract serves as the foundational layer of the protocol, where users can directly stake and unstake their uWatt tokens.

- **Deposit and Withdrawal Management:** Users interact with the `stake(...)` and `unstake(...)` functions to manage their principal. The contract actively tracks the `totalStaked` balance across the system and individual user balances within the `stakedBalance` mapping.
- **Access Control:** The system includes a whitelisting mechanism managed by the `WHITELIST_MANAGER_ROLE`. Unless the `whitelistBypassed` flag is active, only approved addresses are permitted to deposit funds.
- **Rewarder Notification:** To ensure accurate yield distribution, the **Rewarder** contract acts as the source of truth for user balances. Upon any change to a user's staked principal, the contract automatically invokes the `_notifyRewarder(...)` internal function, which updates the user's state in the active reward distribution module.

4.2 Reward Distribution Modules

The protocol abstracts yield distribution logic into external modules that implement the `IRewarder` interface. This design allows the system to utilize different distribution strategies:

- **Claim Module:** A traditional index-based rewarder that calculates proportional yield based on a user's staked balance and a globally tracked `supplyIndex`. It operates on a pull-based model where users actively execute the `claim(...)` function to transfer accumulated reward tokens to their wallets.
- **SuperfluidAdapter Module:** An advanced integration that leverages the Superfluid protocol to stream rewards continuously. Instead of requiring users to manually claim, this adapter converts the underlying reward tokens into SuperTokens. It then deploys individual Distributor contracts that connect to a General Distribution Agreement (GDA) pool, pushing real-time reward streams to stakers.

4.3 Reward Accrual and Claiming

New yields are introduced into the system through the `reportRewards(...)` function inside the StakingUWatts contract. This function is called by a designated `REWARD_MANAGER_ROLE`.

- **Delta Calculation:** The system calculates a `deltaIndex` using the ratio between the incoming rewards and the current total supply of uWatt tokens. This index is then forwarded to the active reward contract alongside the transferred funds.
- **Index-Based Accrual:** If the Claim module is active, the reported rewards increase the global `fundedRewards` and update the `supplyIndex`. When a user claims, their pending rewards are calculated by multiplying their stake by the difference between the current global index and their personal `userIndex`.
- **Stream-Based Accrual:** If the SuperfluidAdapter is active, the reported rewards are wrapped into SuperTokens and transferred to a new Distributor instance. The adapter calculates a specific flow rate to distribute the tokens over a 30-day period. Users can then withdraw their streaming SuperTokens by calling `claim(...)`, which downgrades them back into the underlying ERC20 tokens and transfers them to the user.

4.4 Auto-Compounding Vault & Strategy

To optimize yield generation, the protocol offers the `UWattsVault4626`, an ERC4626-compliant vault designed to automate the compounding of rewards.

- **Vault Mechanics:** Users deposit their uWatt tokens into the vault and receive share tokens in return. The vault relies on a strategy contract, meaning it does not hold the assets itself but delegates management to a strategy contract.
- **StakingStrategy Integration:** The `StakingStrategy` acts as the active manager for the vault's assets. Upon receiving deposits from the vault, the strategy approves and directly deposits the uWatt tokens into the StakingUWatts contract to begin earning yield.

4.5 The Harvest Mechanism

The compounding lifecycle is executed via the permissionless `harvest(...)` function within the `StakingStrategy` contract. This flow ensures that generated rewards are converted back into principal.

- **Reward Collection and Fees:** The strategy first claims any pending rewards from the `StakingUWatts` contract. A configured `performanceFeeBps` is deducted and sent to the protocol's fee recipient. To incentivize external callers to trigger the harvest, a harvester bounty is paid out to the address executing the transaction.
- **Funds Routing:** The strategy uses an on-chain quoter contract to simulate a swap on a Decentralized Exchange (DEX), such as Uniswap V3, to determine the expected `uWatt` output. It compares this output against a fixed quote from an external `IUWattVendor` contract.
- **Execution and Restaking:** The strategy executes the trade through the route that provides the highest amount of `uWatt` tokens. Finally, the newly acquired `uWatt` tokens are restaked into the core contract, effectively compounding the vault's underlying position and increasing the value of the users' shares.

5 Risk Rating Methodology

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

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

- a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;
- b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;
- c) **Low**: The issue is very complex and requires very specific conditions to be met.

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

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

- a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;
- b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;
- c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

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

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

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

- a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;
- b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;
- c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Blocklist bypass via increaseAllowance and transferFrom

File(s): [contracts/staking_module/UWattsVault4626.sol](#)

Description: The UWattsVault4626 contract implements a blocklist mechanism by overriding the approve, transfer, and transferFrom functions with the onlyNotBlocked modifier. This modifier checks that msg.sender is not in the isBlocked mapping.

However, the contract inherits from OZ's ERC20Upgradeable version 4.9.6 but does not override increaseAllowance or decreaseAllowance. As a result, a blocked user can still modify allowances by calling increaseAllowance (which calls the internal _approve function and bypasses the overridden approve check).

Furthermore, the transferFrom override only enforces the onlyNotBlocked check on msg.sender (the spender). It does not validate the status of the from or to addresses.

```
1 function transferFrom(  
2     address from,  
3     address to,  
4     uint256 value  
5 )  
6 public  
7 override(ERC20Upgradeable, IERC20Upgradeable)  
8 // @audit-issue Checks msg.sender (spender), but allows transfer if 'from'  
9 // is blocked.  
10 onlyNotBlocked  
11 returns (bool)  
12 {  
13     return super.transferFrom(from, to, value);  
14 }  
15
```

This creates two bypass vectors:

1. **Bypass via new allowance:** A blocked user calls increaseAllowance to authorize a fresh, unblocked wallet. The unblocked wallet then calls transferFrom to move the funds. Since the non-blocked spender is msg.sender in transferFrom, the check passes;
2. **Bypass via existing allowance:** If a user provided an allowance before being blocked, the authorized spender can still move funds because the owner's blocked status is ignored during transferFrom;

Recommendation(s): Consider overriding the _beforeTokenTransfer hook to enforce blocklist checks. Alternatively, modify the internal _transfer and _approve functions to check that the from and to are not blocklisted.

Status: Fixed at commit [71fe9bae6321ed27e3ae1d3904544a3ab0f04002](#)

Update from the client: Replaced per-function onlyNotBlocked modifier overrides on approve, transfer, and transferFrom with a single _beforeTokenTransfer hook that validates from, to, and msg.sender against the blocklist. This closes the bypass vectors through increaseAllowance/ decreaseAllowance and ensures blocked addresses cannot operate as spenders via pre-existing allowances.

6.2 [Low] Sequential fee application in harvest(...) results in lower than expected harvester bounties

File(s): [contracts/staking_module/StakingStrategy.sol](#)

Description: The harvest(...) function in the StakingStrategy contract is responsible for claiming rewards, deducting fees, and auto-compounding the remaining tokens back into the staking protocol. To ensure the system remains sustainable and incentivized, the protocol defines a performanceFeeBps for the treasury and a harvesterBountyBps to reward the caller who triggers the harvest.

According to the setFees(...) function, the total fees are capped at 2000 basis points (20%) of the total rewards, calculated as the sum of both fee components:

```

1  function setFees(
2      uint256 _performanceFeeBps,
3      uint256 _harvesterBountyBps
4  ) external onlyOwner {
5      // @audit Total allowed fees are validated as the sum of both BPS values
6      if (_performanceFeeBps + _harvesterBountyBps > 2000)
7          revert FeesTooHigh(); // max 20% total
8      // ...
9  }
10

```

However, the harvest(...) function applies these fees sequentially rather than in parallel against the original reward balance. Specifically, the performance fee is subtracted first, and the harvester bounty is then calculated based on the `_remaining_` balance.

```

1  // ...
2  // 2. Deduct performance fee
3  uint256 perfFee = (rewardBal * performanceFeeBps) / BPS;
4  if (perfFee > 0) {
5      rewardToken.safeTransfer(feeRecipient, perfFee);
6  }
7
8  // @audit The performance fee is subtracted from the total reward balance
9  uint256 remaining = rewardBal - perfFee;
10
11 // 3. Pay harvester bounty (incentivizes public harvest calls)
12 // @audit-issue The bounty is calculated on the 'remaining' amount instead of 'rewardBal'
13 uint256 bounty = (remaining * harvesterBountyBps) / BPS;
14 if (bounty > 0) {
15     rewardToken.safeTransfer(caller, bounty);
16     remaining -= bounty;
17 }
18 // ...
19

```

This logic means the actual bounty paid to the harvester is $(\text{Reward} * (1 - \text{performanceFee}) * \text{harvesterBounty})$ instead of the expected $(\text{Reward} * \text{harvesterBounty})$. Consequently, the harvester receives less than the percentage defined in `harvesterBountyBps`, which may lead to insufficient incentives for external actors to maintain the vault's compounding frequency, especially when the performance fee is high.

Recommendation(s): Consider calculating both the `perfFee` and the bounty based on the initial `rewardBal` to ensure the distributed amounts match the basis points configured in the contract state.

Status: Fixed at commit [7cc4b87434e7b1596d96438a6aab8633ad30e6f9](#)

Update from the client: Both `perfFee` and bounty are now calculated in parallel using the original `rewardBal` as the reference value. This ensures that the distributed amounts exactly match the basis points configured through `setFees()`.

6.3 [Info] Strategy replacement can lead to loss of funds

File(s): `contracts/staking_module/UWattsVault4626.sol`

Description: The UWattsVault4626 contract delegates asset management to the StakingStrategy contract. The owner can update this strategy reference using the `setStrategy(...)` function.

However, the `setStrategy` function only updates the strategy state variable pointer to the new address without withdrawing the underlying assets from the previous strategy or migrating them to the new one.

```
1 function setStrategy(address _strategy) external onlyOwner {
2     if (_strategy == address(0)) revert ZeroAddress();
3     IStrategy newStrategy = IStrategy(_strategy);
4
5     // Verify strategy asset matches vault asset
6     if (newStrategy.asset() != asset()) revert StrategyAssetMismatch();
7     address oldStrategy = address(strategy);
8
9     // @audit-issue The pointer is updated, but funds remain in 'oldStrategy'.
10    strategy = newStrategy;
11
12    emit StrategyUpdated(oldStrategy, _strategy);
13 }
14
```

Once the strategy variable is overwritten, the UWattsVault4626 loses the ability to interact with the old strategy.

1. The UWattsVault4626 can only call `withdraw` on the address stored in the strategy variable;
2. The StakingStrategy restricts the `withdraw` function to `onlyVault`;
3. The StakingStrategy also explicitly forbids the owner from using `recoverToken` to retrieve the underlying asset;

This leads to a situation where funds held by the old strategy become locked. Furthermore, the UWattsVault4626's `totalAssets()` function immediately queries the `_new_` strategy (which likely has zero balance), causing the share price to crash to zero.

```
1 function totalAssets() public view override returns (uint256) {
2     if (address(strategy) == address(0)) {
3         return 0;
4     }
5     // @audit Queries the NEW empty strategy, ignoring funds in the old one.
6     return strategy.totalUnderlying();
7 }
8
```

Recommendation(s): Consider the above consequences when deciding to update the strategy contract. Before updating the strategy variable, the vault should withdraw the assets from the current strategy, approve the new strategy, and deposit the assets into it.

Status: Fixed at commit [6259b7f1a3336261e469f8f5d820343d18c91270](#)

Update from the client: Future logic upgrades can be performed through proxy upgrades to the strategy implementation itself. Therefore, a post initialization `setStrategy()` function is unnecessary.

The fix removes the public `setStrategy()` function entirely and replaces it with an internal `_setStrategy()` function that is executed exactly once during initialization. The strategy address is now immutable after deployment.

6.4 [Info] The harvest(...) function logic may be inconsistent depending on the Quoter implementation

File(s): [contracts/staking_module/StakingStrategy.sol](#)

Description: The harvest(...) function in the StakingStrategy contract is designed to compound rewards by choosing the most cost-effective source for uWatt tokens between a Decentralized Exchange (DEX) and a designated uWattVendor. To make this decision, it utilizes a quoter to estimate the output from the DEX.

```

1  function harvest(
2      address caller
3  ) external override nonReentrant returns (uint256 uWattGained) {
4      // ...
5      if (remaining > 0) {
6          // @audit The quoter is used here to compare prices
7          uint256 uWattsFromDEX = quoter.quoteExactInput(
8              rewardToAssetPath,
9              remaining
10         );
11         uint256 uWattsFromVendor = uWattVendor.getUWattAmount(
12             remaining,
13             address(this)
14         );
15
16         if (uWattsFromDEX > uWattsFromVendor) {
17             // DEX is cheaper
18             uWattSource = "DEX";
19             // @audit This internal call calls the quoter again to set slippage
20             uint256 minUWattOut = _getMinimumSwapOutput(remaining);
21             // ...
22             IRouter.ExactInputParams memory params = IRouter
23                 .ExactInputParams({
24                     path: rewardToAssetPath,
25                     recipient: address(this),
26                     amountIn: remaining,
27                     amountOutMinimum: minUWattOut
28                 });
29             uWattGained = router.exactInput(params);
30         } else {
31             // ...
32         }
33     }
34     // ...
35 }
36

```

The logic assumes a specific behavior from the IQuoter implementation. Depending on whether the quoter simulates a full swap or returns a spot price, two issues arise:

1. **Simulation-based Quoter:** If the quoter (e.g., Uniswap V3 QuoterV2) simulates a full swap, it returns the exact amountOut expected (excluding slippage from state changes between the quote and execution). In this case, calculating minUWattOut by applying a maxSlippageBps reduction to the already accurate quote might be redundant or incorrectly calibrated, as uWattsFromDEX is already the "best estimate.";
2. **Spot-price-based Quoter:** If the quoter returns a value based on the current spot price without simulating the impact of the swap size (price impact), the comparison if (uWattsFromDEX > uWattsFromVendor) becomes inaccurate. The actual amount received from the DEX after price impact could be significantly lower than uWattsFromVendor, leading the strategy to pick the DEX when the Vendor was actually the cheaper option;

Recommendation(s): Consider re-evaluating the integration with the quoter to ensure it aligns with the specific implementation used (e.g., Uniswap V3 QuoterV2).

Status: Fixed at commits [6de0936fa87d5c52c876009b4e225427bba34379](#) and [767ceccba31c7f2a727e3b62697e93f1a530d540](#)

Update from the client: We agree that the contract previously did not clearly communicate which type of quoter implementation is required. The harvest() comparison between DEX and Vendor output is only valid when backed by a simulation-based quoter (e.g., Uniswap V3 'QuoterV2') that accounts for full price impact. A spot-price quoter would overestimate DEX output and cause the strategy to prefer the DEX even when the Vendor yields more uWatts. To prevent future misconfiguration, explicit documentation has been added.

6.5 [Info] The reportRewards function is incompatible with tokens that revert on zero approvals

File(s): [contracts/staking_module/StakingUWatts.sol](#)

Description: The reportRewards(...) function assumes that all ERC20 tokens will successfully execute an approve(...) call when the amount is set to 0. There is a known category of non-standard ERC20 tokens (weird tokens), such as the older BNB token on Ethereum (0xB8c77482e45F1F44dE1745F52C74426C631bDD52), that revert when their approve(...) function is called with a 0 value. This category of ERC20s can not be used as staking rewards

Recommendation(s): For this category of tokens, the allowance should be set directly to the new value.

Status: Fixed at commit [a7787bc44b76429f3543294dc4a4a5d6deb23f37](#) and [03c89eb5bacd29f527a4e9405887685d6bb4efc3](#)

Update from the client: We've replaced it with **forceApprove**, which handles edge cases.

6.6 [Info] Unused code

File(s): [contracts/staking_module/SuperfluidAdapter.sol](#), [contracts/staking_module/StakingUWatts.sol](#)

Description: In the SuperfluidAdapter contract, the following errors are unused:

```
1 error InvalidGDAForwarder();
2 error UnitsAlreadySynced();
3 error PoolNotConfigured();
```

As well as the _downgradeTo function:

```
1 function _downgradeTo(address _to, uint256 _amount) internal {
2     if (_to == address(0)) revert ZeroAddressNotAllowed();
3     superfluidToken.downgradeTo(_to, _amount);
4 }
```

Also, the treasury state variable inside the StakingUWatts seems to be unused. It is only set during initialization, but no function calls or uses the treasury.

Recommendation(s): Consider removing the dead code to improve readability.

Status: Fixed at commit [09c1a8e99db19d1088512a0106ba1693ec932b9e](#)

Update from the client: The code has been removed.

6.7 [Best Practices] Redundant calls to the quoter contract in harvest(...) increase gas consumption

File(s): [contracts/staking_module/StakingStrategy.sol](#)

Description: The harvest(...) function in the StakingStrategy contract is designed to collect rewards, determine the most cost-effective source (DEX vs. Vendor) to acquire assetToken, and compound those rewards. When evaluating the DEX as a potential source, the contract utilizes a quoter contract (Uniswap V3 QuoterV2) to simulate the swap and determine the expected output.

Currently, if the DEX is identified as the better option, the contract calls _getMinimumSwapOutput(...) to calculate the slippage protection. This internal function performs a second, identical call to quoter.quoteExactInput(...) with the same parameters already used in the main harvest(...) logic.

As noted in the Uniswap documentation, Quoter contracts are intended primarily for off-chain estimation because they perform a full state-simulated swap, which is computationally expensive and gas-intensive when executed on-chain. Calling this function twice in a single transaction significantly increases the gas cost for the harvester.

```

1  function harvest(...) external override nonReentrant returns (uint256 uWattGained) {
2      // ...
3      if (remaining > 0) {
4          // @audit First call to the quoter to compare sources
5          uint256 uWattsFromDEX = quoter.quoteExactInput(
6              rewardToAssetPath,
7              remaining
8          );
9          uint256 uWattsFromVendor = uWattVendor.getUWattAmount(
10             remaining,
11             address(this)
12         );
13
14         if (uWattsFromDEX > uWattsFromVendor) {
15             uWattSource = "DEX";
16             // @audit Second call to the quoter inside this function
17             uint256 minUWattOut = _getMinimumSwapOutput(remaining);
18             // ...
19         }
20     }
21     // ...
22 }
23

```

The redundant call occurs here:

```

1  function _getMinimumSwapOutput(
2      uint256 rewardAmount
3  ) internal returns (uint256) {
4      // @audit-issue Redundant quoter call; the value was already fetched in harvest()
5      uint256 expectedUWattOut = quoter.quoteExactInput(
6          rewardToAssetPath,
7          rewardAmount
8      );
9
10     return (expectedUWattOut * (BPS - maxSlippageBps)) / BPS;
11 }
12

```

Recommendation(s): Consider refactoring the _getMinimumSwapOutput(...) function to accept the expectedUWattOut as an input parameter instead of fetching it again. This allows the harvest(...) function to reuse the result from the first quoter call, significantly reducing the gas overhead of the compounding process.

Status: Fixed at commit [73990301c360d66a2c48188d30222eadc5638ba4](#)

Update from the client: The redundant second call to quoter.quoteExactInput inside _getMinimumSwapOutput has been removed. The previously obtained uWattsFromDEX value is now reused directly to calculate the slippage floor, eliminating both the inconsistency risk between quotes and the unnecessary gas overhead.

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 Suno Protocol's contracts documentation

The Suno Protocol's team provided an overview of the system components during the kick-off call with a detailed explanation of the intended functionalities and use cases of the system. Moreover, the team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

Remarks about the Suno Protocol's test suite

Initially, the Suno Protocol lacked a formal test suite. During the course of this engagement, the team developed and shared a comprehensive testing framework that addresses the core logic and fund flows of the Staking and Vault modules. We have reviewed this updated suite and note that it provides a foundational layer of security.

The new tests successfully exercise full user journeys, including deposits, withdrawals, and the auto-compounding harvest logic. While the addition of these tests significantly improves confidence and helps verify the mitigations for issues identified in this report, we recommend the team continues to maintain and expand this suite as the codebase evolves to ensure long-term stability and regression testing.

8.1 Automated Tools

8.1.1 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>.

9 About Nethermind

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

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

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

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

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

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

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

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

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

General Advisory to Clients

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

Disclaimer

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