Security Review Report
NM-0563 DCA.fun

NETHERMIND
SECURITY

(August 15, 2025)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 Order Creation and Funding . . . 4
4.2 Order Execution by Fillers . . . 4
4.3 Asset Settlement and Fund Sourcing . . . 4
4.4 Order Cancellation and Asset Withdrawal . . . 4

5 Risk Rating Methodology **5**

6 Issues **6**
6.1 [Medium] Incorrect price scaling formula results in user value loss . . . 6
6.2 [Medium] User can bypass yield fee on staked `tokenOut` . . . 8
6.3 [Medium] Vault state is inconsistent during `fillOrder` callback execution . . . 9
6.4 [Info] Aave market pauses can lead to unfavorable fills due to increasing scaling factor . . . 10
6.5 [Info] High slippage orders that are past the scaling interval cannot be filled . . . 12
6.6 [Info] Leftover `Permit2` allowances of the callback contract can be hijacked by malicious fillers . . . 13
6.7 [Info] Not all fields returned by the Chainlink Verifier Proxy are checked . . . 15
6.8 [Info] The `verifyReportBulk(...)` function can be used by third-parties . . . 16
6.9 [Best Practices] Unsafe downcasting from `uint256` to `uint160` in `_fillOrderPhase2(...)` . . . 17

7 Documentation Evaluation **18**

8 Test Suite Evaluation **19**
8.1 Tests Output . . . 19
8.2 Automated Tools . . . 24
8.2.1 AuditAgent . . . 24

9 About Nethermind **25**

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for DCA.fun. **DCA.fun** is a decentralized infrastructure protocol designed to provide users with the tools to automate Dollar Cost Averaging (DCA) investment strategies on-chain. The project's primary goal is to offer a non-custodial and trust-minimized alternative to centralized platforms, allowing users to execute systematic investment plans while retaining full control over their assets.

The protocol's architecture is built on a two-sided model, connecting users who create investment orders with a permissionless network of executors, known as Fillers. Fillers are economically incentivized to monitor and trigger the execution of these orders. For security and asset segregation, each user's order is assigned its own isolated smart contract vault, ensuring that funds are never mixed and remain under the sole control of their owner. To facilitate secure and accurate trade execution, the system relies on Chainlink Data Streams for verifiable, high-frequency price data. Furthermore, the protocol enhances capital efficiency by offering an optional integration with the Aave protocol, which allows idle assets held within the vaults to generate yield.

The audit comprises 1747 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 9 points of attention, where three are classified as `Medium`, and six are classified as `Informational` or `Best Practices` severity. The issues are summarized in Fig. 1.

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

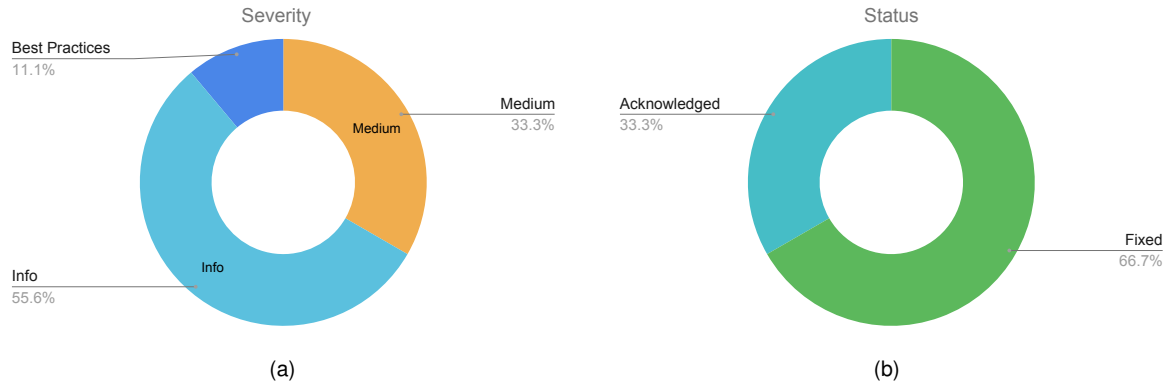

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (3), **Low** (0), **Undetermined** (0), **Informational** (5), **Best Practices** (1). **Distribution of status: Fixed** (6), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	August 12, 2025
Final Report	August 15, 2025
Initial Commit	4dd9891937a6ce2bd807dbbe44da09fbdcf7234b
Final Commit	bcec7763007bcc8392bc90f85bf22fd904238e56
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/dcaDotFun/DcaVaultFactory.sol	64	19	29.7%	13	96
2	src/dcaDotFun/DcaDotFun.sol	527	162	30.7%	143	832
3	src/dcaDotFun/DcaVault.sol	215	89	41.4%	78	382
4	src/dcaDotFun/interfaces/IDcaDotFun.sol	170	167	98.2%	48	385
5	src/dcaDotFun/interfaces/IDcaVaultFactory.sol	27	33	122.2%	8	68
6	src/dcaDotFun/interfaces/IDcaVault.sol	63	94	149.2%	27	184
7	src/dcaDotFun/interfaces/IFillOrderCallback.sol	4	6	150.0%	1	11
8	src/dotFun/DotFun.sol	327	162	49.5%	95	584
9	src/dotFun/libraries/AaveConstants.sol	36	2	5.6%	12	50
10	src/dotFun/interfaces/IDotFun.sol	86	98	114.0%	27	211
11	src/interfaces/IWETH.sol	9	15	166.7%	4	28
12	src/verifierDotFun/VerifierDotFun.sol	88	37	42.0%	24	149
13	src/verifierDotFun/libraries/VerifierLib.sol	58	11	19.0%	10	79
14	src/verifierDotFun/interfaces/IFeeManager.sol	13	11	84.6%	5	29
15	src/verifierDotFun/interfaces/IVerifierProxy.sol	14	15	107.1%	4	33
16	src/verifierDotFun/interfaces/IVerifierdotFun.sol	46	37	80.4%	7	90
	Total	**1747**	**958**	**54.8%**	**506**	**3211**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect price scaling formula results in user value loss	Medium	Fixed
2	User can bypass yield fee on staked `tokenOut`	Medium	Fixed
3	Vault state is inconsistent during `fillOrder` callback execution	Medium	Fixed
4	Aave market pauses can lead to unfavorable fills due to increasing scaling factor	Info	Acknowledged
5	High slippage orders that are past the scaling interval cannot be filled	Info	Fixed
6	Leftover `Permit2` allowances of the callback contract can be hijacked by malicious fillers	Info	Acknowledged
7	Not all fields returned by the Chainlink Verifier Proxy are checked	Info	Acknowledged
8	The `verifyReportBulk(...)` function can be used by third-parties	Info	Fixed
9	Unsafe downcasting from `uint256` to `uint160` in `_fillOrderPhase2(...)`	Best Practices	Fixed

4 System Overview

The DCA.fun protocol is a decentralized application designed to facilitate Dollar Cost Averaging (DCA) investment strategies for users on-chain. It allows users to create orders that are automatically executed over time by a network of independent actors known as Fillers. The system's architecture prioritizes security and asset isolation by creating a unique, non-custodial smart contract vault for each individual DCA order. The protocol integrates with Chainlink Data Streams for reliable price information and optionally with the Aave protocol to allow users to earn yield on their deposited assets while an order is active.

4.1 Order Creation and Funding

The primary entry point for a user is the `DcaDotFun` contract. A user initiates a DCA strategy by calling either `createOrder` for ERC20 tokens or `createOrderNative` for the chain's native asset. When creating an order, the user specifies several parameters, including the input token (`tokenIn`), the output token (`tokenOut`), the amount of `tokenIn` to be spent in each execution period (`spendAmount`), the total number of times the order should repeat (`repeats`), and the time between executions (`freqInterval`).

Upon the creation of an order, the `DcaDotFun` contract invokes the `DcaVaultFactory`, which deploys a new, unique `DcaVault` contract for that specific order. The total amount of `tokenIn` required for all executions (`spendAmount * repeats`) is transferred from the user into this newly created vault. This design ensures that each order's funds are segregated and controlled by the order creator. Users also have the option to stake their assets by setting the `stakeAssetIn` or `stakeAssetOut` flags. If `stakeAssetIn` is enabled, the deposited `tokenIn` is immediately supplied to the Aave protocol from within the `DcaVault` to begin generating yield.

4.2 Order Execution by Fillers

Orders are executed by third-party actors called Fillers, who are incentivized to monitor the protocol for executable orders. An order becomes eligible for execution once the time since its last run exceeds its specified `freqInterval`. To execute an order, a Filler must first fetch the latest price data for the order's token pair from an off-chain Chainlink Data Stream endpoint.

The Filler then calls one of the `fillOrder` functions on the `DcaDotFun` contract, providing the unverified price report as part of the transaction data. The protocol verifies the report's authenticity on-chain by passing it to the `VerifierDotFun` contract, which communicates with Chainlink's `VerifierProxy`. The protocol's accounting logic is designed to work specifically with price data reported with 18 decimals of precision; the introduction of feeds with other decimal formats could lead to incorrect calculations.

To make unfilled orders more attractive over time, the protocol employs a `scalingFactor` tied to the user-defined `slippage`. This mechanism functions like a Dutch auction. Initially, when an order becomes fillable, the effective execution price is set in favor of the order creator by an amount equivalent to their specified `slippage`. As time passes within the order's `scalingInterval`, the `scalingFactor` increases, gradually eroding the creator's price advantage. Halfway through the interval, the price adjustment is neutralized, bringing the effective execution price closer to the market rate. Beyond this point, the price becomes progressively more favorable for the Filler, reaching a maximum discount at the end of the interval. This increases the potential profit for the Filler and encourages timely execution.

4.3 Asset Settlement and Fund Sourcing

Once an order's parameters and price data have been validated, the protocol proceeds with the settlement of assets. The `DcaVault` releases the specified `spendAmount` of `tokenIn` to a recipient address designated by the Filler. In return, the Filler is responsible for providing the corresponding amount of `tokenOut` to the `DcaVault`. All incoming token transfers from the Filler are handled through the `Permit2` contract.

The protocol offers two distinct flows for Fillers to source the required `tokenOut`:

- **Standard Flow**: In the simpler `fillOrder` variant, the transaction's `msg.sender` is assumed to be the source of the `tokenOut` funds. The `Permit2` contract is instructed to pull tokens directly from the Filler's address.
- **Callback Flow**: A more flexible `fillOrder` variant allows the Filler to specify a `callback` contract address. This contract is called during execution, allowing Fillers to implement arbitrary on-chain logic for sourcing the necessary funds (e.g., from a decentralized exchange or a private liquidity pool). In this flow, the specified `callback` address is designated as the source for the `Permit2` transfer.

If the order creator enabled the `stakeAssetOut` option, the incoming `tokenOut` is automatically supplied to the Aave protocol to generate yield.

4.4 Order Cancellation and Asset Withdrawal

The creator of an order retains full control and can cancel it at any time by calling `cancelOrders` on the `DcaDotFun` contract. This action prevents any future fills of the order and marks the associated `DcaVault` as cancelled. Once cancelled, the order's creator (the owner of the vault) can withdraw all remaining assets from their `DcaVault`. This includes any unused `tokenIn` and the full balance of the acquired `tokenOut`. If the assets were staked in Aave, the withdrawal process also claims any yield that was generated. The protocol is designed to take a percentage of this earned yield, as defined by the `yieldSplit` parameter set at the time of order creation.

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] Incorrect price scaling formula results in user value loss

File(s): src/dcaDotFun/DcaDotFun.sol, src/dotFun/DotFun.sol

Description: The protocol implements a dynamic pricing mechanism for order fills, designed to incentivize timely execution. When a user creates an order, they specify a `slippage` percentage. The intended logic is that the fill price starts at a premium over the market price, equal to the `slippage`. Over a `scalingInterval`, this premium linearly decreases, eventually becoming a discount. At the end of the interval, the fill price should be the market price discounted by the `slippage`.

However, the implementation of this price scaling contains a mathematical flaw. The logic first inflates the `tokenInPrice` by the full `slippage` amount. It then calculates a `scalingFactor` that linearly increases from `0` to `2 * slippage` over the `scalingInterval`, and uses this factor to discount the already-inflated price.

```
function _fillOrderValidation(
    uint256 orderId_,
    bytes[] memory unverifiedReports_
) internal returns (OrderValidation memory orderValidationData) {
    // ...
    (uint256 tokenInPrice, uint256 tokenOutPrice) = _getPrices(
        // ...
    );
    // @audit-issue 1. The market price is inflated by the full slippage amount.
    tokenInPrice = tokenInPrice + ((tokenInPrice * order.slippage) / BASIS_POINTS);

    uint256 scalingFactor = _calculateScalingFactor(
        order.slippage,
        order.freqInterval,
        order.scalingInterval,
        order.lastRun
    );

    // @audit The inflated price is passed to the calculation function.
    uint256 amountOfTokenOut = _determineTokenOutAmount(
        order.tokenIn,
        order.tokenOut,
        tokenInPrice,
        tokenOutPrice,
        fillableAmount,
        scalingFactor
    );
    // ...
}
```

The `_determineTokenOutAmount` function in the `DotFun` base contract then applies this `scalingFactor` as a discount to the inflated price.

```
function _determineTokenOutAmount(
    address tokenIn_,
    address tokenOut_,
    uint256 tokenInPrice_,
    uint256 tokenOutPrice_,
    uint256 fillableAmount_,
    uint256 scalingFactor_
) internal view returns (uint256) {
    // ...
    // @audit-issue 2. The inflated price is then discounted by the scaling factor.
    uint256 amountOfTokenOut = (fillableAmount_ *
        tokenInPrice_ *
        (BASIS_POINTS - scalingFactor_) *
        (10 ** outDecimals)) / ((10 ** inDecimals) * BASIS_POINTS * tokenOutPrice_);
    // ...
    return amountOfTokenOut;
}
```

The combination of these two steps leads to an incorrect final price. Let `s` be `slippage / BASIS_POINTS`. The effective price factor becomes `(1 + s) * (1 - scalingFactor)`, where `scalingFactor` (as a fraction of `BASIS_POINTS`) ranges from `0` to `2s`. At the end of the interval, the factor is `(1 + s) * (1 - 2s) = 1 - s - 2s^2`. The intended factor was `1 - s`.

The flawed formula causes the user's `tokenIn` to be sold at a larger discount than specified at the end of the scaling period. This results in the user receiving fewer `tokenOut` tokens than they should, leading to a direct loss of value. The magnitude of this loss is proportional to `2 * (slippage / 10000)^2`, meaning it becomes more significant with higher slippage values. For example, with a 5% slippage, the user experiences an additional 0.5% loss at the end of the scaling interval compared to the intended discounted price. This undermines the predictability and fairness of the pricing mechanism for the protocol's users.

Recommendation(s): Consider revising the price scaling logic to correctly implement the linear interpolation from a premium to a discount.

Status: Fixed.

Update from the client: Commit Hash: `03630bd4522c273dabd4b5ead7a47420abe8012d`

We acknowledge this finding and have successfully reproduced the mathematical error at scale. Our initial testing during development focused on slippage values 1% (100bps), which masked the quadratic error term that becomes significant at higher slippage percentages. We have implemented a corrected linear scaling formula that properly transitions from a premium (market price + slippage) to a discount (market price - slippage) over the scaling interval. The new implementation:

- Consolidates the pricing calculations into a single function to prevent compounding errors;
- Eliminates the unintended 2×slippage² loss at the end of the scaling period;
- Has been validated across a comprehensive range of slippage values (0.01% to 20%) and asset values;
- Correctly produces the intended discount factor of (1 - slippage) at interval completion;

The revised formula ensures users receive the expected number of tokens based on their specified slippage tolerance, addressing the fairness and predictability concerns raised in the audit.

6.2 [Medium] User can bypass yield fee on staked `tokenOut`

File(s): src/dcaDotFun/DcaVault.sol

Description: Users can create DCA orders that stake the received `tokenOut` in Aave by setting the `stakeAssetOut` flag to `true`. When these orders are filled, the corresponding `DcaVault` contract receives `aTokenOut`. The principal amount of `tokenOut` acquired from the fill is tracked in the `escrow` state variable, while the vault's actual balance of `aTokenOut` increases as it accrues yield.

The protocol is designed to collect a fee on this generated yield, determined by the `yieldSplit` parameter. This fee collection is intended to happen within the `_handleTokenWithdrawal(...)` function when an order is cancelled. This function calculates the yield by taking the difference between the `aTokenOut` balance and the `escrow` amount.

However, the `DcaVault` contract includes an `onlyOwner` function, `withdrawErc20(...)`, which allows the creator to call this function with the `tokenOut` or `aTokenOut` address before the order is cancelled and withdraw the principal amount tracked by `escrow`. This function also resets the `escrow` variable to `0` without processing the accrued yield or collecting the protocol's fee.

```
function withdrawErc20(address token_) external onlyOwner {
    // ...
    if (balance == 0) revert NoTokensToWithdraw();
    if ((token_ == tokenOut || token_ == aTokenOut) && !isCancelled) {
        // ...
        // @audit-issue User can withdraw the principal amount (`escrow`) before the order is cancelled.
        uint256 eligableAmount = escrow > balance ? balance : escrow;
        // @audit-issue The `escrow` state variable is reset to zero,
        // while the yield remains in the contract.
        escrow = 0;

        if (token_ == aTokenOut) {
            if (eligableAmount > _maxAssetsWithdrawableFromAave(tokenOut)) revert WithdrawExceedsMax();
            IPool(pool).withdraw(tokenOut, eligableAmount, recipient);
        } else {
            IERC20(token_).safeTransfer(recipient, eligableAmount);
        }
        emit WithdrawErc20(orderId, token_, recipient, eligableAmount);
    } else {
        IERC20(token_).safeTransfer(recipient, balance);
        emit WithdrawErc20(orderId, token_, recipient, balance);
    }
}
```

This allows a malicious user to bypass the fee collection mechanism entirely. After withdrawing the principal and zeroing out `escrow`, the user can cancel the order. During cancellation, the logic that handles `tokenOut` and collects the yield fee is skipped because it is conditioned on `escrow > 0`.

```
function cancelOrder(uint256 repeats_) external onlyFunContract whenNotCancelled {
    isCancelled = true;
    (bool tokenInStaked, bool tokenOutStaked) = _stakedAssets();

    _handleTokenWithdrawal(tokenIn, aTokenIn, tokenInStaked, spendAmount * repeats_, 0);
    // @audit Because `escrow` was set to 0, this check fails and the fee collection logic is bypassed.
    if (escrow > 0) {
        uint256 escrowedAmount = escrow;
        escrow = 0;
        _handleTokenWithdrawal(tokenOut, aTokenOut, tokenOutStaked, 0, escrowedAmount);
    }
}
```

As a result, the user can then call `withdrawErc20(...)` a second time after the order is cancelled. This call will fall into the `else` branch, transferring the entire remaining `aTokenOut` balance—which represents the untaxed yield—to the recipient, thus stealing part of the protocol's intended revenue.

Recommendation(s): Consider accounting for possible yield on the `tokenOut` even if `escrow` is zero.

Status: Fixed.

Update from the client Commit Hash: `52525b3769aa425908e9f336170e2dee3af96de9`

The main fix was removing the conditional escrow check in the `cancelOrder()` function, ensuring that yield fee calculation always occurs during order cancellation, regardless of whether a user previously called `withdrawErc20()` to reset the escrow.This prevents users from bypassing the protocol's yield fee collection mechanism and stealing the protocol's intended revenue from Aave yield generation.

Before the fix: A malicious user could:

- Call `withdrawErc20(aTokenOut)` → resets escrow to 0;
- Call `cancelOrder()` → skips yield fee calculation due to escrow == 0;

- Call `withdrawErc20(aTokenOut)` again → withdraws remaining yield without fees;

After the fix:

- Even if `withdrawErc20(aTokenOut)` resets `escrow` to 0;
- `cancelOrder()` will still call `_handleTokenWithdrawal()` with the original escrow amount;
- Yield fees are properly calculated and collected based on the actual aToken balance vs escrow amount;
- The protocol receives its intended fee share;
- The changes to `_handleTokenWithdrawal()`:;
- Explicit Token Targeting: aToken == aTokenOut ensures yield fee logic only applies to the correct token
- Escrow Bypass Prevention: Removing escrowAmount > 0 check means fees are collected even if escrow was maliciously reset to 0
- Complete Yield Capture: When escrowAmount = 0 (after malicious withdrawal), the entire balance is treated as yield, ensuring fee collection

6.3 [Medium] Vault state is inconsistent during `fillOrder` callback execution

File(s): src/dcaDotFun/DcaDotFun.sol, src/dcaDotFun/DcaVault.sol

Description: The `fillOrder(...)` function that accepts a `callback` parameter splits its execution into two phases around an external call. The first phase, `_fillOrderPhase1(...)`, updates the `DcaVault`'s state, while the second phase, `_fillOrderPhase2(...)`, handles the transfer of assets. The `escrow` state variable in the `DcaVault` contract is designed to track the principal amount of `tokenOut` held within the vault, which is particularly important for calculating yield on staked assets.

A temporal desynchronization exists between the vault's internal accounting (`escrow`) and its actual asset balance. The `_fillOrderPhase1(...)` function calls `DcaVault.fillOrder(...)`, which immediately increments the `escrow` variable. However, the corresponding `tokenOut` assets are not transferred into the vault until `_fillOrderPhase2(...)`, which executes *after* the external `callback` has completed.

```
function fillOrder(
    bytes calldata encodedData_,
    address callback_,
    address recipient_,
    bytes calldata data_
) external whenOrderFillNotPaused nonReentrant {
    (
        // ...
    // @audit-issue The vault's `escrow` is updated here, before tokens are transferred.
    ) = _fillOrderPhase1(encodedData_, recipient_);

    // @audit An external call is made while the vault's state is inconsistent.
    if (!IFillOrderCallback(callback_).fillOrderCallback(data_)) revert FillOrderCallbackFailed();

    // @audit The tokens corresponding to the `escrow` update are only transferred here.
    _fillOrderPhase2(orderId, order, callback_, fee, amountOfTokenOutMinusFee, orderValidationData, recipient_);
}
```

This creates a window of time during the callback's execution where the vault's state is inconsistent. The `escrow` variable reflects tokens that the vault has not yet received, meaning its internal accounting does not match its on-chain balance.

Recommendation(s): Consider refactoring the `fillOrder(...)` function to ensure that state changes only occur after all necessary conditions, including asset transfers, are met.

Status: Fixed.

Update from the client: Commit Hash: `6540f76daad11ca1de4b3ca2b1737465dd163f74`

We acknowledge the identified temporal desynchronization issue where the vault's escrow variable was updated in `_fillOrderPhase1` before the corresponding tokens were transferred in `_fillOrderPhase2`, creating an inconsistent state window during callback execution.

Solution Implemented:

Removed escrow update from `_fillOrderPhase1`

Created a new function in DcaVault with onlyFunContract modifier

Moved escrow update to `_fillOrderPhase2`, executing after token transfers but before order finalization.

This ensures the vault's internal accounting (escrow) always matches its actual token balance.

The solution has been implemented and validated through comprehensive testing. The vault state now remains consistent throughout the entire order filling process, eliminating the vulnerability window.

6.4 [Info] Aave market pauses can lead to unfavorable fills due to increasing scaling factor

File(s): src/dcaDotFun/DcaDotFun.sol, src/dotFun/DotFun.sol

Description: Users can create DCA orders where the output tokens (`tokenOut`) are automatically staked in Aave to generate yield. The protocol also features a `scalingFactor` mechanism, which creates a dutch auction over time. If an order is not filled promptly, the `scalingFactor` increases, making the order progressively more attractive for fillers by reducing the amount of `tokenOut` they need to provide.

The issue arises if an Aave market for a staked `tokenOut` is paused by Aave governance, which has occurred in the past. When an Aave market is paused, the `_maxAssetsSuppliableToAave(...)` function in the `DotFun` contract will return `0`.

```
function _maxAssetsSuppliableToAave(
    DataTypes.ReserveData memory reserveData_
) internal view returns (uint256) {
    uint256 reserveConfigMap = reserveData_.configuration.data;
    // ...
    // @audit If a market is paused, this condition is met and the function returns 0.
    if (
        (reserveConfigMap & ~AAVE_ACTIVE_MASK == 0)
            || (reserveConfigMap & ~AAVE_FROZEN_MASK != 0)
            || (reserveConfigMap & ~AAVE_PAUSED_MASK != 0)
    ) {
        return 0;
    }
    // ...
}
```

Consequently, any attempt to fill the order will revert in the `_fillOrderPhase2(...)` function of the `DcaDotFun` contract, as the amount to be supplied will be greater than the allowed `0`.

```
function _fillOrderPhase2(
    // ...
) internal {
    // ...
    if (order_.stakeAssetOut) {
        // @audit-issue If the Aave market for tokenOut is paused, _maxAssetsSuppliableToAave(...)
        // returns 0, causing this check to revert with DepositExceedsMax.
        if (
            amountOfTokenOutMinusFee_
                >= _maxAssetsSuppliableToAave(
                    IPool(aavePool).getReserveData(order_.tokenOut)
                )
        ) revert DepositExceedsMax();
        // ...
    }
    // ...
}
```

While the order is unfillable due to the paused Aave market, the `_calculateScalingFactor(...)` function continues to increase the `scalingFactor` based on `block.timestamp`.

```
function _calculateScalingFactor(
    uint256 slippage_,
    uint256 freqInterval_,
    uint256 scalingInterval_,
    uint256 lastRun_
) internal view returns (uint256 scalingFactor) {
    // @audit elapsedTime increases with block.timestamp, regardless of fillability.
    uint256 elapsedTime = block.timestamp > (lastRun_ + freqInterval_)
        ? block.timestamp - (lastRun_ + freqInterval_)
        : 0;

    if (elapsedTime >= scalingInterval_) {
        scalingFactor = slippage_ * 2;
    } else {
        scalingFactor = (elapsedTime * slippage_ * 2) / scalingInterval_;
    }
}
```

This means that once the Aave market is unpaused and the order becomes fillable again, the user will receive a significantly worse exchange rate. The `scalingFactor` will have increased during the entire downtime, penalizing the user for an external event beyond anyone's control. While users define a maximum `slippage`, they may not be aware that it can be fully reached due to Aave market conditions.

Recommendation(s): Consider documenting this behavior to ensure users understand the risks associated with staking `tokenOut` in Aave. Alternatively, a mechanism could be implemented to pause the `scalingFactor` increase for an order if its corresponding Aave market is paused.

Status: Acknowledged.

Update from the client: The concern regarding scaling factor increases during Aave market pauses will be addressed through documentation. When Aave markets are paused, orders with staked tokenIn become unfillable while the scaling factor continues to increase upto the user-defined slippage.

6.5 [Info] High slippage orders that are past the scaling interval cannot be filled

File(s): src/dcaDotFun/DcaDotFun.sol, src/dotFun/DotFun.sol

Description: When a filler executes a Dollar Cost Averaging (DCA) order, the protocol calculates a `scalingFactor`. This factor is designed to make an order more attractive to fillers over time if it remains unfilled, effectively creating a Dutch auction. The attractiveness is increased by adjusting the required `amountOfTokenOut` that the filler must provide.

The problem arises from the combination of the maximum allowed `slippage` and how the `scalingFactor` is calculated and applied. The `DcaDotFun` constructor allows `_maxSlippage` to be set as high as `10_000`, which is equal to `BASIS_POINTS`.

When an order is being filled, the `_calculateScalingFactor(...)` function in the `DotFun` contract determines the scaling value. If the time elapsed since the order was ready to be filled (`lastRun_ + freqInterval_`) exceeds the order's `scalingInterval_`, the `scalingFactor` is set to `slippage_ * 2`.

```
function _calculateScalingFactor(
    uint256 slippage_,
    uint256 freqInterval_,
    uint256 scalingInterval_,
    uint256 lastRun_
) internal view returns (uint256 scalingFactor) {
    uint256 elapsedTime = block.timestamp > (lastRun_ + freqInterval_)
        ? block.timestamp - (lastRun_ + freqInterval_)
        : 0;

    // @audit If we're past the scaling interval, use max scaling (2x slippage).
    if (elapsedTime >= scalingInterval_) {
        scalingFactor = slippage_ * 2;
    } else {
        // ...
    }
}
```

This `scalingFactor` is then used in the `_determineTokenOutAmount(...)` function to calculate the final output amount. The calculation involves subtracting the `scalingFactor_` from `BASIS_POINTS`, which effectively decreases the amount of `tokenOut` that the filler has to provide, which makes the order more attractive.

```
function _determineTokenOutAmount(
    address tokenIn_,
    address tokenOut_,
    uint256 tokenInPrice_,
    uint256 tokenOutPrice_,
    uint256 fillableAmount_,
    uint256 scalingFactor_
) internal view returns (uint256) {
    // ...
    // @audit-issue This subtraction will underflow if scalingFactor_ > BASIS_POINTS.
    uint256 amountOfTokenOut = (
        fillableAmount_ * tokenInPrice_ * (BASIS_POINTS - scalingFactor_)
            * (10 ** outDecimals)
    ) / ((10 ** inDecimals) * BASIS_POINTS * tokenOutPrice_);

    if (amountOfTokenOut < 1) revert InvalidTokenOutAmount();

    return amountOfTokenOut;
}
```

If a user creates an order with a `slippage` greater than `5_000`, and the order is not filled before its `scalingInterval` passes, the `scalingFactor` will be set to a value greater than `10_000`. This will cause an arithmetic underflow in `_determineTokenOutAmount(...)`, making any attempt to fill the order revert. Consequently, the order becomes permanently unfillable, and the creator's funds remain locked in the vault until the order is manually cancelled.

Recommendation(s): Consider constraining the maximum allowed `slippage` during order creation. A safer upper bound for `_maxSlippage` could be `5_000` (i.e., `BASIS_POINTS / 2`). This would ensure that the maximum possible `scalingFactor` (`2 * slippage`) does not exceed `BASIS_POINTS`, preventing the underflow and ensuring all orders remain fillable regardless of how long they have been active.

Status: Fixed.

Update from the client: Commit Hash: `8d0cd63d50264dc04e302c0207262990d5de7a9b`

We have implemented the auditor's recommendation by adding validation checks to constrain the maximum allowed slippage to `5,000` basis points (50%). Calidation in the `constructor` to ensure `_maxSlippage < 5,000 (BASIS_POINTS / 2)`, the same validation was added to `setMinMaxSlippage` to prevent runtime updates that could exceed this limit.

6.6 [Info] Leftover `Permit2` allowances of the callback contract can be hijacked by malicious fillers

File(s): src/dcaDotFun/DcaDotFun.sol

Description: The protocol allows any party, known as a "filler", to execute Decentralized Dollar Cost Averaging (DCA) orders on behalf of users. The `fillOrder(...)` function exists in two variants. One variant assumes the `msg.sender` provides the necessary `tokenOut` to fill the order. The second, more flexible variant, allows the caller to specify an arbitrary `callback_` address. This callback contract is intended to implement custom logic for sourcing the required `tokenOut`.

A potential issue arises because the caller of `fillOrder(...)` can specify an arbitrary `callback_` address as the source of funds, while also controlling the `recipient_` address that receives the swapped `tokenIn`.

```
function fillOrder(
    bytes calldata encodedData_,
    address callback_,
    address recipient_,
    bytes calldata data_
) external whenOrderFillNotPaused nonReentrant {
    // ...
    // @audit The recipient_ gets the token IN.
    (/*...*/) = _fillOrderPhase1(encodedData_, recipient_);
    // @audit The specified callback is invoked.
    if (!IFillOrderCallback(callback_).fillOrderCallback(data_)) {
        revert FillOrderCallbackFailed();
    }

    _fillOrderPhase2(
        orderId,
        order,
        callback_, // @audit The callback address is passed as the source of funds.
        fee,
        amountOfTokenOutMinusFee,
        orderValidationData,
        recipient_
    );
}
```

The `_fillOrderPhase2(...)` function then uses the provided `callback_` address as the `filler_` from which to pull `tokenOut` via `Permit2`.

```
function _fillOrderPhase2(
    uint256 orderId_,
    Order memory order_,
    address filler_,
    uint256 fee_,
    uint256 amountOfTokenOutMinusFee_,
    // ...
) internal {
    // @audit-issue Funds are pulled from `filler_`, which is the `callback_` address from the caller.
    IPermit2(_permit2).transferFrom(
        filler_, feeCollector, uint160(fee_), order_.tokenOut
    );

    if (order_.stakeAssetOut) {
        // ...
        IPermit2(_permit2).transferFrom(
            filler_,
            address(this),
            uint160(amountOfTokenOutMinusFee_),
            order_.tokenOut
        );
        // ...
    } else {
        IPermit2(_permit2).transferFrom(
            filler_,
            order_.vault,
            uint160(amountOfTokenOutMinusFee_),
            order_.tokenOut
        );
    }
    // ...
}
```

A malicious actor can exploit this design. They can call `fillOrder(...)` specifying another filler's callback contract as the `callback_` and their own address as the `recipient_`. If the victim's callback contract has an active `Permit2` approval to the `DcaDotFun` contract, the attacker can use the victim's funds (`tokenOut`) to fill the order, while the reward (`tokenIn`) is sent to the attacker's address.

This attack vector is external to the `DcaDotFun` protocol itself but represents an integration risk for developers building filler bots. Without proper access control in their callback contracts, their funds are at risk of being drained by any other user of the `DcaDotFun` protocol.

Recommendation(s): While this is primarily an integration risk, the protocol can take steps to mitigate it. One approach to mitigate this at the protocol level would be to use `msg.sender` as the source of funds for the `Permit2.transferFrom(...)` call. This would ensure that the entity initiating the transaction is also the one providing the funds, whether it's an Externally Owned Account (EOA) or a filler's callback contract.

Status: Acknowledged.

Update from the client: The protocol intentionally allows arbitrary callback addresses to optimize for gas efficiency and composability. This design enables fillers to implement sophisticated routing strategies, leverage flash loans, and aggregate liquidity across multiple sources without requiring intermediate transfers. Using `msg.sender` as the fund source would necessitate an additional transfer (callback → msg.sender → protocol), increasing gas costs, making fills less profitable.

Some not all items to consider as **advanced** filler:

- Use separate contracts for callback logic and fund storage;
- Implement just-in-time approvals with amount limits;
- Add caller verification in callback contracts;
- Use time-boxed or amount-limited Permit2 approvals;
- Consider using a proxy pattern where the callback contract has minimal approvals;

6.7 [Info] Not all fields returned by the Chainlink Verifier Proxy are checked

File(s): src/dcaDotFun/DcaDotFun.sol

Description: The `_getPrice(...)` and `_getPrices(...)` functions in the `DotFun` contract are responsible for obtaining verified token prices from Chainlink Data Streams. They call the `verifyReportBulk(...)` function on the `VerifierDotFun` contract, which in turn queries Chainlink's Verifier Proxy.

The current implementation validates that the price report is not stale by checking the `observationsTimestamp` against the configured `maxFeedAge_`. It also ensures the `feedId` matches the expected token's feed to prevent using an incorrect report.

```
function _getPrice(
    bytes[] calldata unverifiedReports_,
    bytes32 tokenFeedId_,
    uint256 maxFeedAge_
) internal returns (uint256 tokenPrice) {
    CustomReport[] memory decodedReport = IVerifierDotFun(verifierDotFun)
        .verifyReportBulk(unverifiedReports_, 1);

    CustomReport memory tokenReport = decodedReport[0];

    uint256 currentTimestamp = block.timestamp;

    // @audit Only the observationsTimestamp is checked for staleness.
    if (tokenReport.observationsTimestamp + maxFeedAge_ < currentTimestamp)
    {
        revert ExpiredReport();
    }

    // @audit-issue The marketStatus and validFromTimestamp fields are not checked.
    if (tokenReport.feedId != tokenFeedId_) revert FeedIdMismatch();

    if (tokenReport.price < 1) revert PriceIsZero();

    return (uint256(int256(tokenReport.price)));
}
```

However, the `CustomReport` struct returned by the verifier contains other fields, such as `validFromTimestamp` and `marketStatus`, that are currently not utilized or checked. For the currently supported crypto asset data streams, `validFromTimestamp` is, in practice, the same as `observationsTimestamp`, and the `marketStatus` is consistently "Open" (a value of `2`). While this makes additional checks seem unnecessary at present, this behavior is reported because reliance on these unchecked, externally-provided values introduces a degree of third-party risk. The internal workings of Chainlink's data streams may change in the future, potentially affecting the validity of these assumptions.

The protocol's `VerifierLib` is designed to handle different report schemas (v3 for crypto and v4 for RWAs), which suggests that support for Real World Assets (RWAs) may be intended in the future. For RWA data streams, the `marketStatus` field will be important, as it indicates whether a market is open for trading. Executing an order based on a price from a closed market could lead to value loss or other unexpected behavior. This issue serves as an informational notice for future development cycles.

Recommendation(s): Extra care should be taken by the administrators when adding new data feeds via the `setTokenProps(...)` function, especially if expanding support to include non-crypto assets like RWAs, to ensure all relevant report fields are handled appropriately by the protocol.

Status: Acknowledged.

Update from the client: The protocol's architecture is designed with a shared VerifierDotFun contract that serves as infrastructure for all products under the DotFun umbrella. DotFun acts as a base contract that handles core price validation (including timestamp validation), while inheriting contracts are responsible for validating any additional fields they require. This separation allows different products to implement their specific validation needs without modifying shared infrastructure.

We acknowledge the importance of validating all external data fields, including `marketStatus` and `validFromTimestamp`, regardless of our trust in Chainlink as a data provider. While these fields have predictable values for crypto assets (24/7 markets with marketStatus always "open"), inheriting contracts will implement appropriate validation for any fields they utilize. This approach ensures defensive programming practices are maintained while preserving the flexibility to support diverse asset types and use cases across future products that inherit from DotFun.

6.8 [Info] The `verifyReportBulk(...)` function can be used by third-parties

File(s): src/verifierDotFun/VerifierDotFun.sol

Description: The `VerifierDotFun` contract is responsible for verifying the authenticity of Chainlink Data Stream reports. The price verification is handled by the `verifyReportBulk(...)` function, which calls the Chainlink `VerifierProxy` to confirm that a given price report was signed by the Decentralized Oracle Network.

The `verifyReportBulk(...)` function is declared as `external` but lacks any access control mechanism.

```
function verifyReportBulk(
    bytes[] memory unverifiedReports,
    uint256 reportLength
)
    // @audit-issue The function is external and has no access control.
    external
    virtual
    override
    returns (CustomReport[] memory decodedReport)
{
    if (unverifiedReports.length != reportLength) {
        revert InvalidReportLength();
    }

    bytes[] memory verifiedReportData = s_verifierProxy.verifyBulk(
        unverifiedReports, abi.encode(s_feeTokenAddress)
    );

    decodedReport = VerifierLib.decodeReportBulk(verifiedReportData);
}
```

This means that any external user or contract can call this function. This introduces two primary risks:

1. **Abuse of Whitelisted Status**: The protocol is whitelisted by Chainlink to use their Data Stream verification service without paying a fee. A third party can abuse this by using the `VerifierDotFun` contract as a free on-chain verification oracle for their own, unrelated applications;
2. **Fee Draining** / **Denial of Service**: If the protocol's whitelisted status is ever revoked, it will be required to pay for verification in `LINK` tokens from the `VerifierDotFun` contract's balance. A malicious actor could repeatedly call `verifyReportBulk(...)` to grief the protocol and deplete its `LINK` token funds. This would cause a Denial of Service, as core functions in `DcaDotFun` like `createOrder(...)` and `fillOrder(...)` would begin to fail when they are unable to pay for price verification.

Recommendation(s): Consider implementing access control for the `verifyReportBulk(...)` function. Access should be restricted to only trusted contracts within the protocol ecosystem that require its services, such as the main `DcaDotFun` contract.

Status: Fixed.

Update from the client: Commit Hash: `4710357ef6f54d5899242b19d98de0456034e6bb`

Our contracts don't hold LINK tokens or pay Oracle fees directly. Chainlink confirmed that third-party abuse of our whitelisted status is not a concern under our arrangement. Despite no vulnerability, we implemented access control as best practice:

- Added `onlyFun` modifier to `verifyReportBulk()`;
- Added `setFunContract()` for managing authorized callers;

6.9 [Best Practices] Unsafe downcasting from `uint256` to `uint160` in `_fillOrderPhase2(...)`

File(s): src/dcaDotFun/DcaDotFun.sol

Description: The `_fillOrderPhase2(...)` function is the second phase of filling an order, where a filler provides `tokenOut` to complete the swap. The function utilizes Uniswap's `Permit2` contract to handle token transfers from the filler.

The problem is that the `Permit2` contract's `transferFrom(...)` function accepts amounts as `uint160`, while the `_fillOrderPhase2(...)` function receives the `fee_` and `amountOfTokenOutMinusFee_` as `uint256`. The code performs a direct, unsafe cast from `uint256` to `uint160`.

According to Uniswap's security advisory for `Permit2`, this unsafe downcast can lead to the value being truncated if it exceeds the maximum value for a `uint160`. While this is unlikely with standard, regular-decimal tokens, it could become an issue for high-decimal "memecoins". Since the DCA protocol aims to support a wide variety of assets, this could lead to unexpected behavior where a smaller amount than intended is transferred from the filler.

```
function _fillOrderPhase2(
    uint256 orderId_,
    Order memory order_,
    address filler_,
    uint256 fee_,
    uint256 amountOfTokenOutMinusFee_,
    OrderValidation memory orderValidationData_,
    address recipient_
) internal {
    // @audit The `fee_` variable is unsafely cast from uint256 to uint160.
    // While the risk is lower as fees are a fraction of the total
    // amount, it is still best practice to use safe casting.
    IPermit2(_permit2).transferFrom(
        filler_, feeCollector, uint160(fee_), order_.tokenOut
    );

    if (order_.stakeAssetOut) {
        if (
            amountOfTokenOutMinusFee_
                >= _maxAssetsSuppliableToAave(
                    IPool(aavePool).getReserveData(order_.tokenOut)
                )
        ) revert DepositExceedsMax();

        // @audit-issue The `amountOfTokenOutMinusFee_` is unsafely cast.
        IPermit2(_permit2).transferFrom(
            filler_,
            address(this),
            uint160(amountOfTokenOutMinusFee_),
            order_.tokenOut
        );
        // ...
    } else {
        // @audit-issue The `amountOfTokenOutMinusFee_` is unsafely cast.
        IPermit2(_permit2).transferFrom(
            filler_,
            order_.vault,
            uint160(amountOfTokenOutMinusFee_),
            order_.tokenOut
        );
    }
    // ...
}
```

Recommendation(s): Consider using a safe casting library, such as OpenZeppelin's `SafeCast`, to handle the downcasting from `uint256` to `uint160`. This approach aligns with the best practices recommended by Uniswap for integrating with `Permit2`.

Status: Fixed.

Update from the client: Commit Hash: `c375a496701e6681937d9a0bd0c4294b4f0a84d9`

We've implemented the recommended fix using OpenZeppelin's SafeCast library to handle the `uint256` to `uint160` conversions in `_fillOrderPhase2(...)`.

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 DCA.fun's documentation

The **DCA.fun** team provided comprehensive documentation for the protocol. The codebase is well-documented with detailed NatSpec comments, and the project's `README.md` file offers a clear overview of the system's architecture, core components, and user flows. Furthermore, the team was highly responsive and available for internal sync calls, promptly addressing all questions and discussion points raised by the Nethermind Security team. This collaborative approach provided valuable insights and ensured a thorough understanding of the protocol's technical aspects throughout the engagement.

8 Test Suite Evaluation

8.1 Tests Output

```
Compiler run successful with warnings:
Warning (5667): Unused function parameter. Remove or comment out the variable name to
silence this warning.
--> test/protocol/dcaVault/helpers/VaultHelpers.sol:42:9:
|
42 | uint256 _escrowAmount
| ^^^^^^^^^^^^^^^^^^^^^
Ran 1 test for
test/protocol/dcaVaultFactoty/dcaVaultFactory_edgeCasesAndFailures.t.sol:DcaVaultFactoryEd
geCasesAndFailures
[PASS] test_dcaVaultFactory_vault_double_create() (gas: 290690183)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.01ms (128.50µs CPU time)
Ran 1 test for
test/protocol/dcaVaultFactoty/dcaVaultFactory_constructorAndInitialization.t.sol:DcaVaultFactor
yConstructorAndInitializer
[PASS] test_dcaVaultFactory_constructor_initialization() (gas: 18963)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.06ms (58.33µs CPU time)
Ran 1 test for
test/protocol/dcaVaultFactoty/dcaVaultFactory_eventEmission.t.sol:DcaVaultFactoryEventEmiss
ion
[PASS] test_dcaVaultFactory_createVault_event_emission() (gas: 374808)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 488.63µs (66.71µs CPU time)
Ran 2 tests for
test/protocol/dcaVaultFactoty/dcaVaultFactory_createVault.t.sol:DcaVaultFactoryEventEmission
[PASS] test_dcaVaultFactory_createVault_event_emission() (gas: 374787)
[PASS] test_dcaVaultFactory_createVault_success() (gas: 392920)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 616.13µs (161.08µs CPU time)
Ran 5 tests for test/protocol/dcaVault/dcaVault_accessControl.t.sol:DcaVaultAccessControl
[PASS] test_dcaVault_cancelOrder_access_control() (gas: 21793)
[PASS] test_dcaVault_fillOrder_access_control() (gas: 18111)
[PASS] test_dcaVault_updateEscrow_access_control() (gas: 15911)
[PASS] test_dcaVault_withdrawErc20_access_control() (gas: 18270)
[PASS] test_dcaVault_withdrawNative_access_control() (gas: 15971)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 564.79µs (79.46µs CPU time)
Ran 3 tests for
test/protocol/dcaVaultFactoty/dcaVaultFactory_deterministicAddress.t.sol:DcaVaultFactoryDeter
ministicAddress
[PASS] test_dcaVaultFactory_predictVaultAddress_accuracy() (gas: 369392)
[PASS] test_dcaVaultFactory_vault_address_determinism() (gas: 9845)
[PASS] test_dcaVaultFactory_vault_address_uniqueness() (gas: 9858)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 521.08µs (78.83µs CPU time)
Ran 3 tests for test/protocol/dcaVault/dcaVault_cancelOrder.t.sol:DcaVaultCancelOrder
[PASS] test_dcaVault_cancelOrder_already_cancelled() (gas: 419823)
[PASS] test_dcaVault_cancelOrder_no_yield() (gas: 427049)
[PASS] test_dcaVault_cancelOrder_with_escrow() (gas: 565219)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 1.09ms (505.29µs CPU time)
Ran 2 tests for test/protocol/ValidateSetup.t.sol:CreateOrder
[PASS] test_validate_Setup() (gas: 3340)
[PASS] test_validate_token_props() (gas: 46662)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 7.86s (237.92µs CPU time)
Ran 8 tests for test/protocol/dcaDotFun/dcaDotFun_constructor.t.sol:DcaDotFunConstructor
[PASS] test_dcaDotFun_constructor_invalidYieldSplit() (gas: 324652)
[PASS] test_dcaDotFun_constructor_max_gt_10000_InvalidSlippage() (gas: 324660)
[PASS] test_dcaDotFun_constructor_min_gt_max_InvalidSlippage() (gas: 327567)
[PASS] test_dcaDotFun_constructor_zeroAddress_permit2() (gas: 319666)
[PASS] test_dcaDotFun_constructor_zeroAddress_vaultFactory() (gas: 319644)
[PASS] test_dcaDotFun_constructor_zeroAddress_wrappedNative() (gas: 319687)
[PASS] test_dotFun_constructor_zeroAddress_feeCollector() (gas: 88548)
[PASS] test_dotFun_constructor_zeroAddress_verifierDotFun() (gas: 88527)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 789.58µs (553.21µs CPU time)
Ran 28 tests for
test/protocol/dcaDotFun/dcaDotFun_tokenManagement.t.sol:DcaDotFunTokenManagement
[PASS] test_dcaDotFun_getTokenProps() (gas: 58460)
[PASS] test_dcaDotFun_pauseCreateOrder() (gas: 36208)
[PASS] test_dcaDotFun_pauseFillOrder() (gas: 36209)
```

```
[PASS] test_dcaDotFun_setAavePool1() (gas: 28581)
[PASS] test_dcaDotFun_setAavePool_AavePoolAlreadySet() (gas: 28950)
[PASS] test_dcaDotFun_setFeeCollector() (gas: 29255)
[PASS] test_dcaDotFun_setFeeCollector_ZeroAddress() (gas: 13546)
[PASS] test_dcaDotFun_setMaxFeedAgeCreateOrder() (gas: 46779)
[PASS] test_dcaDotFun_setMaxFeedAgeFillOrder() (gas: 46709)
[PASS] test_dcaDotFun_setMaxScalingInterval() (gas: 46713)
[PASS] test_dcaDotFun_setMinExecutionValue() (gas: 37148)
[PASS] test_dcaDotFun_setMinMaxSlippage() (gas: 50978)
[PASS] test_dcaDotFun_setMinMaxSlippage_max_gt_1000InvalidSlippage() (gas: 13563)
[PASS] test_dcaDotFun_setMinMaxSlippage_min_gt_max_InvalidSlippage() (gas: 13497)
[PASS] test_dcaDotFun_setMinOrderFrequencyInterval() (gas: 46733)
[PASS] test_dcaDotFun_setProtocolFee() (gas: 34457)
[PASS] test_dcaDotFun_setTimestampTolerance() (gas: 32080)
[PASS] test_dcaDotFun_setTokenProps_1() (gas: 219827)
[PASS] test_dcaDotFun_setTokenProps_AccessControlUnauthorizedAccount() (gas: 22500)
[PASS] test_dcaDotFun_setTokenProps_InvalidAaveAsset() (gas: 65459)
[PASS] test_dcaDotFun_setTokenProps_ZeroFeed() (gas: 21638)
[PASS] test_dcaDotFun_setTokenState() (gas: 70113)
[PASS] test_dcaDotFun_setTokenState_AccessControlUnauthorizedAccount() (gas: 20061)
[PASS] test_dcaDotFun_setVaultFactory() (gas: 46920)
[PASS] test_dcaDotFun_setVaultFactory_ZeroAddress() (gas: 13513)
[PASS] test_dcaDotFun_setVerifierDotFun() (gas: 34609)
[PASS] test_dcaDotFun_setYieldSplit() (gas: 46793)
[PASS] test_dcaDotFun_setYieldSplit_InvalidYieldSplit() (gas: 13466)
Suite result: ok. 28 passed; 0 failed; 0 skipped; finished in 9.16s (1.30s CPU time)
Ran 1 test for
test/protocol/dcaVaultFactoty/dcaVaultFactory_aavePoolManagement.t.sol:DcaVaultFactoryAav
ePoolManagement
[PASS] test_dcaVaultFactory_setAavePool_success() (gas: 726413)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 550.54µs (94.54µs CPU time)
Ran 3 tests for
test/protocol/dcaVaultFactoty/dcaVaultFactory_accessControl.t.sol:DcaVaultFactoryAccessCont
rol
[PASS] test_dcaVaultFactory_createVault_access_control() (gas: 381034)
[PASS] test_dcaVaultFactory_ownership_transfer() (gas: 387509)
[PASS] test_dcaVaultFactory_setAavePool_access_control() (gas: 24527)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 612.58µs (182.17µs CPU time)
Ran 1 test for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_notStaked_staked.t.sol:CreateOr
derNotStakedTokenInStakedTokenOut
[PASS] test_dcaDotFun_createOrder_notStaked_tokenIn_staked_tokenOut() (gas: 953523)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.20s (2.34s CPU time)
Ran 1 test for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_notStaked_notStaked.t.sol:Creat
eOrderNotStakedTokenInNotStakedTokenOut
[PASS] test_dcaDotFun_createOrder_notStaked_tokenIn_notStaked_tokenOut() (gas: 904155)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.20s (2.34s CPU time)
Ran 1 test for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_notStaked_native.t.sol:CreateOr
derNotStakedNative
[PASS] test_dcaDotFun_createOrder_notStaked_native() (gas: 898675)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.20s (2.34s CPU time)
Ran 5 tests for
test/protocol/dcaDotFun/dcaDotFun_accessControl.t.sol:DcaDotFunAccessControl
[PASS] test_dcaDotFun_access_control_events() (gas: 137213)
[PASS] test_dcaDotFun_admin_functions_onlyOwner() (gas: 78967)
[PASS] test_dcaDotFun_ownership_transfer() (gas: 102474)
[PASS] test_dcaDotFun_pause_blocks_operations() (gas: 1352638)
[PASS] test_dcaDotFun_role_based_access() (gas: 127101)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 10.33s (2.47s CPU time)
Ran 3 tests for test/protocol/dcaVault/dcaVault_eventEmission.t.sol:DcaVaultEventEmission
[PASS] test_dcaVault_cancelOrder_event() (gas: 850433)
[PASS] test_dcaVault_withdrawErc20_event() (gas: 978534)
[PASS] test_dcaVault_withdrawNative_event() (gas: 867810)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 10.32s (2.47s CPU time)
Ran 1 test for test/protocol/dcaVault/dcaVault_initialization.t.sol:DcaVaultInitialization
[PASS] test_dcaVault_initialize_already_initialized() (gas: 395302)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 511.29µs (52.58µs CPU time)
Ran 4 tests for test/protocol/dcaDotFun/cancelOrder/dcaDotFun_cancelOrder.t.sol:CancelOrder
[PASS] test_dcaDotFun_cancelOrders_NotOrderCreator() (gas: 861737)
[PASS] test_dcaDotFun_cancelOrders_OrderCancelled() (gas: 853459)
```

```
[PASS] test_dcaDotFun_cancelOrders_repeats_set_to_0() (gas: 853084)
[PASS] test_dcaDotFun_cancelOrders_with_fill_escrow_set_to_0() (gas: 1264555)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 10.33s (2.47s CPU time)
Ran 4 tests for
test/protocol/dcaDotFun/cancelOrder/dcaDotFun_cancelOrder_notStaked_notStaked.t.sol:Canc
elOrderNotStakedTokenInNotStakedTokenOut
[PASS]
test_dcaDotFun_cancelOrder_notStaked_tokenIn_notStaked_tokenOut_no_fills_no_yield()
(gas: 966426)
[PASS]
test_dcaDotFun_cancelOrder_notStaked_tokenIn_notStaked_tokenOut_no_fills_with_yield()
(gas: 232)
[PASS]
test_dcaDotFun_cancelOrder_notStaked_tokenIn_notStaked_tokenOut_with_fill_no_yield()
(gas: 1404442)
[PASS]
test_dcaDotFun_cancelOrder_notStaked_tokenIn_notStaked_tokenOut_with_fill_with_yield()
(gas: 231)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 11.22s (3.36s CPU time)
Ran 1 test for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_staked_notStaked.t.sol:CreateOr
derStakedTokenInNotStakedTokenOut
[PASS] test_dcaDotFun_createOrder_staked_tokenIn_notStaked_tokenOut() (gas: 1127798)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 12.44s (4.59s CPU time)
Ran 4 tests for
test/protocol/dcaDotFun/cancelOrder/dcaDotFun_cancelOrder_staked_notStaked.t.sol:CancelO
rderStakedTokenInNotStakedTokenOut
[PASS] test_dcaDotFun_cancelOrder_staked_tokenIn_notStaked_tokenOut_no_fills_no_yield()
(gas: 1213301)
[PASS]
test_dcaDotFun_cancelOrder_staked_tokenIn_notStaked_tokenOut_no_fills_with_yield() (gas:
1313169)
[PASS] test_dcaDotFun_cancelOrder_staked_tokenIn_notStaked_tokenOut_with_fill_no_yield()
(gas: 186)
[PASS]
test_dcaDotFun_cancelOrder_staked_tokenIn_notStaked_tokenOut_with_fill_with_yield() (gas:
1844903)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 12.96s (5.10s CPU time)
Ran 3 tests for
test/protocol/dcaDotFun/fillOrder/dcaDotFun_fillOrder_usdc_weth_staked.t.sol:FillOrderUsdcW
ethNotStaked
[PASS] test_dcaDotFun_fillOrder_not_staked_tokenIn_staked_tokenOut() (gas: 2034348)
[PASS] test_dcaDotFun_fillOrder_staked_tokenIn_not_staked_tokenOut() (gas: 2103449)
[PASS] test_dcaDotFun_fillOrder_staked_tokenIn_staked_tokenOut() (gas: 2432561)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 13.19s (5.33s CPU time)
Ran 4 tests for
test/protocol/dcaDotFun/fillOrder/dcaDotFun_scalingSlippage.t.sol:ScalingSlippage
[PASS] test_dcaDotFun_scaling_slippage_at_half_of_scalingFactor() (gas: 1069531)
[PASS] test_dcaDotFun_scaling_slippage_freqInterval_max() (gas: 1069442)
[PASS] test_dcaDotFun_scaling_slippage_freqInterval_min() (gas: 1069008)
[PASS] test_dcaDotFun_scaling_slippage_gt_half_of_scalingFactor() (gas: 1069556)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 7.45s (1.78s CPU time)
Ran 14 tests for test/protocol/dcaDotFun/fillOrder/dcaDotFun_quote_revert.t.sol:QuoteRevert
[PASS] test_dcaDotFun_quote_FeedIdMismatch() (gas: 1064029)
[PASS] test_dcaDotFun_quote_InvalidReportLength() (gas: 1242762)
[PASS] test_dcaDotFun_quote_InvalidTokenOutAmount() (gas: 991961)
[PASS] test_dcaDotFun_quote_OrderDoesNotExist() (gas: 149117)
[PASS] test_dcaDotFun_quote_OrderNotFillable() (gas: 971082)
[PASS] test_dcaDotFun_quote_PriceIsZero() (gas: 1023975)
[PASS] test_dcaDotFun_quote_PriorToOrderExecution() (gas: 969841)
[PASS] test_dcaDotFun_quote_TimeStampMismatch() (gas: 991599)
[PASS] test_dcaDotFun_quote_TokenInNotActive() (gas: 981180)
[PASS] test_dcaDotFun_quote_TokenOutNotActive() (gas: 981404)
[PASS] test_dcaDotFun_quote_cancel_order_OrderNotActive() (gas: 961908)
[PASS] test_dcaDotFun_quote_no_repeats_remaining_OrderNotActive() (gas: 1276145)
[PASS] test_dcaDotFun_quote_tokenIn_ExpiredReport() (gas: 1272748)
[PASS] test_dcaDotFun_quote_tokenOut_ExpiredReport() (gas: 1277504)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 7.68s (2.01s CPU time)
Ran 7 tests for
test/protocol/dcaDotFun/dcaDotFun_orderManagement.t.sol:DcaDotFunOrderManagement
[PASS] test_dcaDotFun_createOrderNative_success() (gas: 898675)
[PASS] test_dcaDotFun_createOrder_invalid_amounts() (gas: 434810)
[PASS] test_dcaDotFun_createOrder_invalid_tokens() (gas: 206395)
[PASS] test_dcaDotFun_createOrder_success() (gas: 904109)
```

```
[PASS] test_dcaDotFun_getOrders_by_token_pair() (gas: 2282606)
[PASS] test_dcaDotFun_getOrders_by_user() (gas: 2473535)
[PASS] test_dcaDotFun_order_id_generation() (gas: 2485003)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 9.36s (2.53s CPU time)
Ran 26 tests for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_revert.t.sol:CreateOrderRevert
[PASS] test_dcaDotFun_OrderDoesNotExist() (gas: 14131)
[PASS] test_dcaDotFun_createOrder_CreateOrderPaused() (gas: 877217)
[PASS] test_dcaDotFun_createOrder_ExpiredReport() (gas: 278856)
[PASS] test_dcaDotFun_createOrder_FeedIdMismatch() (gas: 272391)
[PASS] test_dcaDotFun_createOrder_InsufficientAllowance() (gas: 139077)
[PASS] test_dcaDotFun_createOrder_InsufficientBalance() (gas: 170647)
[PASS] test_dcaDotFun_createOrder_InvalidFrequencyInterval() (gas: 163825)
[PASS] test_dcaDotFun_createOrder_InvalidReportLength() (gas: 227960)
[PASS] test_dcaDotFun_createOrder_InvalidSlippage_1() (gas: 159569)
[PASS] test_dcaDotFun_createOrder_InvalidSlippage_2() (gas: 161712)
[PASS] test_dcaDotFun_createOrder_MinExecutionValue() (gas: 287178)
[PASS] test_dcaDotFun_createOrder_PriceIsZero() (gas: 175485)
[PASS] test_dcaDotFun_createOrder_RecipientIsZeroAddress() (gas: 146139)
[PASS] test_dcaDotFun_createOrder_RepeatsIsZero() (gas: 123587)
[PASS] test_dcaDotFun_createOrder_TokenInAndOutSame() (gas: 150502)
[PASS] test_dcaDotFun_createOrder_TokenInNotActive() (gas: 161873)
[PASS] test_dcaDotFun_createOrder_TokenOutNotActive() (gas: 164012)
[PASS] test_dcaDotFun_createOrder_notStaked_tokenIn_staked_tokenOut_InvalidAaveAsset()
(gas: 1329659)
[PASS]
test_dcaDotFun_createOrder_notStaked_tokenIn_staked_tokenOut_TokenNotStakable() (gas:
170219)
[PASS]
test_dcaDotFun_createOrder_scalingFactort_gt_half_of_freqInterval_InvalidScalingFactor()
(gas: 161980)
[PASS]
test_dcaDotFun_createOrder_scalingFactort_gt_maxScalingFactor_InvalidScalingFactor()
(gas: 163961)
[PASS] test_dcaDotFun_createOrder_staked_tokenIn_TokenNotStakable() (gas: 158255)
[PASS] test_dcaDotFun_createOrder_staked_tokenIn_staked_tokenOut_DepositExceedsMax()
(gas: 443736)
[PASS] test_dcaDotFun_createOrder_staked_tokenIn_staked_tokenOut_InvalidAaveAsset()
(gas: 1368538)
[PASS] test_dcaDotFun_createOrder_staked_tokenIn_staked_tokenOut_TokenNotStakable()
(gas: 170241)
[PASS] test_dcaDotFun_getOrderTokens() (gas: 861487)
Suite result: ok. 26 passed; 0 failed; 0 skipped; finished in 8.80s (3.00s CPU time)
Ran 5 tests for
test/protocol/dcaDotFun/cancelOrder/dcaDotFun_cancelOrder_notStaked_staked.t.sol:CancelO
rderNotStakedTokenInStakedTokenOut
[PASS]
test_dcaDotFun_cancelOrder_notStaked_tokenIn_staked_tokenOut_cancelled_add_pre_cance
l() (gas: 1304237)
[PASS] test_dcaDotFun_cancelOrder_notStaked_tokenIn_staked_tokenOut_no_fill_with_yield()
(gas: 233)
[PASS] test_dcaDotFun_cancelOrder_notStaked_tokenIn_staked_tokenOut_no_fills_no_yield()
(gas: 231)
[PASS] test_dcaDotFun_cancelOrder_notStaked_tokenIn_staked_tokenOut_with_fill_no_yield()
(gas: 1662081)
[PASS]
test_dcaDotFun_cancelOrder_notStaked_tokenIn_staked_tokenOut_with_fill_with_yield() (gas:
1756233)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 10.09s (4.29s CPU time)
Ran 1 test for test/protocol/dcaVault/dcaVault_withdrawNative.t.sol:DcaVaultWithdrawNative
[PASS] test_dcaVault_withdrawNative_success() (gas: 418311)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 535.75µs (64.54µs CPU time)
Ran 21 tests for test/protocol/dcaVault/dcaVault_withdrawErc20.t.sol:DcaVaultWithdrawErc20
[PASS] test_dcaVault_withdrawErc20_aTokenIn_cancelled_add_post_cancel() (gas: 1329470)
[PASS] test_dcaVault_withdrawErc20_aTokenIn_cancelled_add_prior_to_cancel() (gas:
1355234)
[PASS] test_dcaVault_withdrawErc20_aTokenIn_not_cancelled() (gas: 1077426)
[PASS] test_dcaVault_withdrawErc20_aTokenOut() (gas: 1575831)
[PASS]
test_dcaVault_withdrawErc20_aTokenOut_cancel_withdrawErc20_to_bypass_yield_split() (gas:
1703441)
```

```
[PASS] test_dcaVault_withdrawErc20_aTokenOut_cancelled_add_post_cancel() (gas:
1223960)
[PASS] test_dcaVault_withdrawErc20_aTokenOut_not_cancelled_balance_eq_escrow() (gas:
1575794)
[PASS] test_dcaVault_withdrawErc20_aTokenOut_not_cancelled_balance_gt_escrow() (gas:
1685732)
[PASS] test_dcaVault_withdrawErc20_cancelled() (gas: 919993)
[PASS] test_dcaVault_withdrawErc20_non_order_token() (gas: 1693283)
[PASS] test_dcaVault_withdrawErc20_not_cancelled() (gas: 927899)
[PASS] test_dcaVault_withdrawErc20_tokenIn_cancelled_add_post_cancel() (gas: 879050)
[PASS] test_dcaVault_withdrawErc20_tokenIn_cancelled_add_prior_to_cancel() (gas: 870698)
[PASS] test_dcaVault_withdrawErc20_tokenIn_not_cancelled() (gas: 862794)
[PASS] test_dcaVault_withdrawErc20_tokenOut_cancelled_add_post_cancel() (gas: 1284655)
[PASS] test_dcaVault_withdrawErc20_tokenOut_cancelled_add_prior_to_cancel() (gas:
1271383)
[PASS] test_dcaVault_withdrawErc20_tokenOut_not_cancelled_balance_eq_escrow() (gas:
1293932)
[PASS] test_dcaVault_withdrawErc20_tokenOut_not_cancelled_balance_gt_escrow() (gas:
1423549)
[PASS] test_dcaVault_withdrawErc20_tokenOut_with_escrow() (gas: 1443738)
[PASS] test_dcaVault_withdrawErc20_zero_address() (gas: 862414)
[PASS] test_dcaVault_withdrawErc20_zero_balance() (gas: 1636321)
Suite result: ok. 21 passed; 0 failed; 0 skipped; finished in 10.07s (4.50s CPU time)
Ran 6 tests for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder.t.sol:CreateOrderMissingTests
[PASS] test_dcaDotFun_createOrder_IncrementOrderId() (gas: 856152)
[PASS] test_dcaDotFun_createOrder_firstExecution_nonZero() (gas: 881069)
[PASS] test_dcaDotFun_createOrder_multipleOrders() (gas: 1520489)
[PASS] test_dcaDotFun_createOrder_protocolFee() (gas: 872386)
[PASS] test_dcaDotFun_createOrder_recipient() (gas: 868013)
[PASS] test_dcaDotFun_createOrder_yieldSplit() (gas: 1090462)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 13.60s (4.91s CPU time)
Ran 1 test for
test/protocol/dcaDotFun/dcaDotFun_feeManagement.t.sol:DcaDotFunFeeManagement
[PASS] test_dcaDotFun_zero_fee_scenario() (gas: 1285976)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.67s (2.38s CPU time)
Ran 1 test for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_staked_staked.t.sol:CreateOrder
StakedTokenInStakedTokenOut
[PASS] test_dcaDotFun_createOrder_staked_tokenIn_staked_tokenOut() (gas: 1172276)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.30s (4.23s CPU time)
Ran 1 test for test/protocol/dcaDotFun/fillOrder/dcaDotFun_fillOrder_revert.t.sol:FillOrderRevert
[PASS] test_dcaDotFun_fillOrder_FillOrderPaused() (gas: 1316134)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.02s (2.32s CPU time)
Ran 1 test for
test/protocol/verifierDotFun/verifierDotFun_setFunContract.t.sol:VerifierDotFunSetFunContractT
est
[PASS] test_verifierDotFun_setFunContract_onlyOwner() (gas: 35271)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.55s (81.08ms CPU time)
Ran 1 test for
test/protocol/verifierDotFun/verifierDotFun_withdrawToken.t.sol:VerifierDotFunWithdrawTokenT
est
[PASS] test_verifierDotFun_withdrawToken_onlyOwner() (gas: 203184)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.66s (187.56ms CPU time)
Ran 1 test for
test/protocol/dcaDotFun/flashRouter/dcaDotFun_flashRouter_fillOrder.t.sol:FlashRouterFillOrde
r
[PASS] test_dcaDotFun_flashRouter_fillOrder() (gas: 3603659)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 14.26s (5.79s CPU time)
Ran 3 tests for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_native_revert.t.sol:CreateOrderN
ativeRevert
[PASS] test_dcaDotFun_createOrder_native_revert_InvalidEthAmount() (gas: 102255)
[PASS] test_dcaDotFun_createOrder_native_revert_NotZeroAddress() (gas: 122714)
[PASS] test_dcaDotFun_createOrder_native_revert_msgValue_0_MinExecutionValue() (gas:
230853)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 10.15s (1.80s CPU time)
Ran 20 tests for
test/protocol/dcaDotFun/dcaDotFun_eventEmissions.t.sol:DcaDotFunEventEmissions
[PASS] test_dcaDotFun_cancelOrder_event() (gas: 854433)
[PASS] test_dcaDotFun_createOrderNative_event() (gas: 856475)
[PASS] test_dcaDotFun_createOrder_event() (gas: 865957)
[PASS] test_dcaDotFun_fillOrder_event() (gas: 1310240)
```

```
[PASS] test_dcaDotFun_setAavePool_event() (gas: 7866580)
[PASS] test_dcaDotFun_setMaxFeedAgeCreateOrder_event() (gas: 20347)
[PASS] test_dcaDotFun_setMaxFeedAgeFillOrder_event() (gas: 20347)
[PASS] test_dcaDotFun_setMaxScalingInterval_event() (gas: 20325)
[PASS] test_dcaDotFun_setMinMaxSlippage_event() (gas: 25997)
[PASS] test_dcaDotFun_setMinOrderFrequencyInterval_event() (gas: 20390)
[PASS] test_dcaDotFun_setVaultFactory_event() (gas: 20478)
[PASS] test_dcaDotFun_setYieldSplit_event() (gas: 17531)
[PASS] test_dotFun_pauseCreateOrder_event() (gas: 29313)
[PASS] test_dotFun_pauseFillOrder_event() (gas: 29358)
[PASS] test_dotFun_setFeeCollector_event() (gas: 20666)
[PASS] test_dotFun_setMinExecutionValue_event() (gas: 25880)
[PASS] test_dotFun_setProtocolFee_event() (gas: 20418)
[PASS] test_dotFun_setTimestampTolerance_event() (gas: 20367)
[PASS] test_dotFun_setTokenProps_event() (gas: 72320)
[PASS] test_dotFun_setTokenState_event() (gas: 19908)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 9.92s (8.70s CPU time)
Ran 4 tests for
test/protocol/verifierDotFun/verifierDotFun_accessControl.t.sol:VerifierDotFunAccessControlTes
t
[PASS] test_verifierDotFun_setFunContract_NotOwner() (gas: 15678)
[PASS] test_verifierDotFun_setManagersFeeTokenAndApprove_NotOwner() (gas: 13326)
[PASS] test_verifierDotFun_verifyReportBulk_NotFun() (gas: 15705)
[PASS] test_verifierDotFun_withdrawToken_NotOwner() (gas: 13615)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 8.72s (98.39ms CPU time)
Ran 2 tests for
test/protocol/dcaDotFun/fillOrder/dcaDotFun_fillOrder_usdc_weth_notStaked.t.sol:FillOrderUsd
cWethNotStaked
[PASS] test_dcaDotFun_fillOrder_not_staked_slippage_100pct_usdc_weth() (gas: 1651448)
[PASS] test_dcaDotFun_fillOrder_not_staked_slippage_x_usdc_weth() (gas: 1312842)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 9.58s (5.02s CPU time)
Ran 4 tests for
test/protocol/dcaDotFun/cancelOrder/dcaDotFun_cancelOrder_staked_staked.t.sol:CancelOrde
rStakedTokenInStakedTokenOut
[PASS] test_dcaDotFun_cancelOrder_staked_tokenIn_staked_tokenOut_no_fills_no_yield()
(gas: 1264584)
[PASS] test_dcaDotFun_cancelOrder_staked_tokenIn_staked_tokenOut_no_fills_with_yield()
(gas: 1364476)
[PASS] test_dcaDotFun_cancelOrder_staked_tokenIn_staked_tokenOut_with_fill_no_yield()
(gas: 1983391)
[PASS] test_dcaDotFun_cancelOrder_staked_tokenIn_staked_tokenOut_with_fill_with_yield()
(gas: 2103370)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 19.54s (5.92s CPU time)
Ran 1 test for
test/protocol/dcaDotFun/createOrder/dcaDotFun_createOrder_staked_native.t.sol:CreateOrder
StakedNative
[PASS] test_dcaDotFun_createOrder_staked_native() (gas: 1114444)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.66s (3.79s CPU time)
Ran 43 test suites in 30.66s (325.49s CPU time): 210 tests passed, 0 failed, 0 skipped (210
total tests)
```

8.2 Automated Tools

8.2.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. 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.