SMART CONTRACTS REVIEW

zokyo

July 9th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING ELEKTRIK

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 1 High issue: 1 acknowledged = - 4 points deducted
- 6 Medium issues: 6 acknowledged = - 10 points deducted
- 12 Low issues: 9 resolved and 3 acknowledged = - 1 points deducted
- 9 Informational issues: 7 resolved and 2 acknowledged = 0 points deducted

Thus, 100 - 4 - 10 - 1 = 85

TECHNICAL SUMMARY

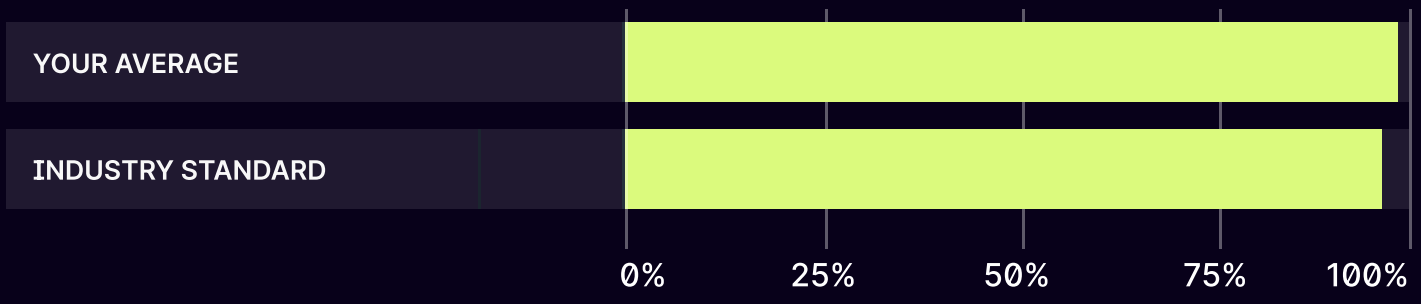
This document outlines the overall security of the Elektrik smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Elektrik smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

Testable Code



94,95% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Elektrik team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Elektrik repository:
1st repository: https://github.com/BlockApex/Elektrik_Airdrops
Initial commit: a8e16e266064598d5c8c3bc17f6c4c6ce59a60d2
Last commit: 081118d934ed08eb5e31633648be02c67b757f6c

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./Airdrops.sol
- ./libraries/ClaimLib.sol

2nd repository: https://github.com/BlockApex/Elektrik-V2-Contracts
Initial commit: f0aa8a71848bb0c98beb4acbc763ce5010c71e51
Last commit: e93d1d7a59f07aa69b9e7ed75348acba5da5644c

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./AdvancedOrderEngineErrors.sol
- ./Predicates.sol
- ./AdvancedOrderEngine.sol
- ./libraries/Decoder.sol
- ./libraries/OrderEngine.sol
- ./Vault.sol
- ./Helper/TargetContract.sol
- ./Helper/VerifyPredicatesLogic.sol
- ./Helper/GenerateCalldata.solLink

3rd repository: https://github.com/BlockApex/Elektrik_Static_Oracle
Initial commit: 2f4f4e2c3025748fc0b704bd6db4fb965b2915fa
Last commit: 178c2f16013c6fbc314c38905ac900bd7dc023c9

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./UsdOracle.sol
- ./StaticOracle.sol
- ./libraries/OracleLibraryPlus.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Elektrik smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract/s logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

Elektrik is a cutting-edge decentralized exchange (DEX) protocol integrated with the Lightlink Network. It enables direct peer-to-peer trading without the need for intermediaries or centralized market makers.

The Elektrik protocol's codebase for the audit consists of contracts for airdrops, limit orders, and static oracle functions.

Elektrik Order Engine →

Elektrik is an order matching service , orders are matched based on specific criteria such as price, size, and time. On the smart contract side (AdvancedOrderEngine.sol) the operator role calls the "fillOrders()" passing an array of orders with their respective sell and buy amount (among other params) . Orders need to be validated first i.e the order should not be an expired one , sell/buy amounts should be non zero , checks to see if the order has already been filled and some other sanity checks are performed. Orders are then processed fully or partially and it is ensured that the limit price of the order is respected , if the order was a partially

fillable order then the filledSellAmount[Orderhash] mapping of the order gets updated accordingly (with the amount filled / executedSellAmount) and the fee is validated such that the minimum fee to process an order is respected.

If the order was a complete fill order then filledSellAmount[] is updated to match the order's original sell token amount and the fee gets validated same as above.

After the successful processing of an order the signature of the order is verified , it is ensured that the order (orderHash) has been signed by the maker address off-chain.

Pre-interactions are optional and are executed if the order's preInteraction length is sufficient to store an address.The calldata and the target contract gets decoded (the target/ interaction address is verified to be correct) and the "fillOrderPreInteraction()" is invoked on the interaction contract.

The processing of the order ends with the sell tokens being received from the maker and fee is sent to the fee collector.

If defined , the facilitator logic gets executed where funds are transferred to the interaction target , and finally the order is finalised by sending the buy tokens to the recipient and post execution logic is executed if defined.

Moreover , the orders can be private , where only the respective operator of that order must be the one calling the fillOrders() function .

Operators can be removed or added by the owner address , moreover the owner maintains a whitelist of tokens i.e. the processing of the order reverts if either the buy token or the sell token is not whitelisted.

The Order Execution Service handles the actual execution of matched orders. It ensures that the orders are executed accurately and in a timely manner. Additionally, the system supports various order types, including Limit Orders, TWAPs (Time-Weighted Average Price), Iceberg Orders, Take Profit, and Stop Loss orders. These order types provide flexibility and customization options for traders.

To facilitate settlement, the system incorporates different methods, such as Dex-settlement and bid-ask on-chain settlement. These methods enable efficient and secure settlement processes for different market conditions. The Settlement Contract and Oracles play important roles in ensuring the accuracy and integrity of the settlement process.

Elektrik Airdrop Mechanism →

Airdrops.sol contract manages the logic to distribute airdrop claims. A merkle tree implementation is used to carry out the airdrop logic , the merkle root gets calculated off-chain with all the participants eligible for the airdrop and then the merkle root gets assigned in the Airdrops.sol smart contract . The user trying to claim the airdrop provides the merkle proof with the amount and it is checked if the user is eligible and the ELTK airdrop gets minted to the user. Openzeppelin Merkle Proof library is being used which makes the system secure and robust.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Elektrik team and the Elektrik team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Facilitators can borrow assets and never repay it back	High	Acknowledged
2	Centralization Risk Due to Overpowered Owner	Medium	Acknowledged
3	Missing Handling for Tokens with Transfer Fees in fillOrders()	Medium	Acknowledged
4	Orders Might Revert Since Fee Is Optional	Medium	Acknowledged
5	filledSellAmount Would Be Incorrect For Fee-On-Transfer Tokens	Medium	Acknowledged
6	getUsdPrice WILL NOT WORK WITH COMPLEX/LARGE PATHS	Medium	Acknowledged
7	TWAP CAN BE MANIPULATED IF THE PERIOD IS SHORT ENOUGH	Medium	Acknowledged
8	USE A MULTISIG ACCOUNT FOR THE OWNER	Low	Acknowledged
9	LACK OF EVENTS	Low	Resolved
10	MISSING SANITY CHECKS FOR IMPORTANT VARIABLES	Low	Resolved
11	Temporary DOS if one order contained a blacklisted address	Low	Acknowledged
12	Owner can renounce ownership	Low	Resolved
13	Improper Order of Modifiers in fillOrders()	Low	Resolved
14	Misleading Documentation for recipient Address Handling	Low	Resolved
15	Missing Sanity checks	Low	Resolved

#	Title	Risk	Status
16	rewardContract and merkleRoot Cannot be updated	Low	Resolved
17	Incorrect Event Emission in Constructor	Low	Resolved
18	Missing Context Header in for merkle proof	Low	Acknowledged
19	Unused Return Value from mint() Function	Low	Resolved
20	Commented-Out Code Results in Unnecessary Event Emission for Unchanged Whitelist Status	Informational	Resolved
21	Uninitialized minimumFee and Ability to Set minimumFee to Zero	Informational	Acknowledged
22	Floating Pragma and Outdated Solidity version	Informational	Resolved
23	Public Function Could Be Marked External	Informational	Resolved
24	Lack of input validation in or() and and() functions	Informational	Acknowledged
25	POOL PAIR IS NOT CHECKED IF IT IS SUPPORTED	Informational	Resolved
26	getUsdPrice DOES NOT RETURN USD PRICE	Informational	Resolved
27	CATCH ARRAY LENGTH	Informational	Resolved
28	USE ++i INSTEAD OF i++	Informational	Resolved

Facilitators can borrow assets and never repay it back

It is mentioned in the documentation that a `Facilitator` has the capability to borrow assets from the vault. However, upon reviewing the `AdvancedOrderEngine` smart contract, no mechanism for verifying whether the borrowed assets have been repaid by the Facilitator is observed.
Without proper checks in place, there is a high possibility of assets being borrowed without repayment, leading to potential loss within the system.

File: AdvancedOrderEngine.sol

```
701:     function _processFacilitatorInteraction(
702:         bytes calldata facilitatorInteraction,
703:         OrderEngine.Order[] calldata orders,
704:         uint256[] calldata executedSellAmounts,
705:         uint256[] calldata executedBuyAmounts,
706:         ERC20[] calldata borrowedTokens,
707:         uint256[] calldata borrowedAmounts
708:     ) private {
..
724:             // Transfer funds to the 'interactionTarget' address.
725:             for (uint256 i; i < borrowedTokens.length; ) {
726:                 _sendAsset(
727:                     borrowedTokens[i],
728:                     borrowedAmounts[i],
729:                     interactionTarget
730:                 );
731:                 unchecked {
732:                     ++i;
733:                 }
734:             }
```

Recommendation:
Add a check to validate that the borrowed assets have been repaid by the Facilitator.

Client comment:
Defense Reason: Design Choice.
DEFENSE: Consider the following example

	User1	User2	User3
SELL	10 USDC	11 USDC	0.0096 ETH
BUY	0.0048 ETH	0.0048 ETH	20 USDC

In this situation, User3's order will satisfy both User1's and User2's orders. Here's how it works:

- User3 sells `0.0096` ETH, which fulfills User1's and User2's requests to buy `0.0048` ETH each.
- In this chain trade, there will be 1 extra USDC token left.

Why does the facilitator get an incentive?

- Facilitator Incentive: Facilitators (bots) are rewarded with the extra token for their service. This incentive mechanism ensures that anyone can create these bots, whitelist them, and start matching orders in our distributed architecture.
- No Unpaid Borrowing: Facilitators cannot borrow assets without repaying. Users will always receive their specified buy amount. Any extra token is an incentive for the facilitator's service.
- Comparison with CoW Swap: On platforms like CoW Swap, solvers (facilitators) can keep the extra token, but there is an additional layer. If the solver willingly foregoes the extra token, their match is accepted, and they are awarded CoW tokens as an incentive.

Auditors comment: We have reviewed the explanation provided by the client and accept that this is the intended design choice.

Centralization Risk Due to Overpowered Owner

The `AdvancedOrderEngine` smart contract grants significant control to the contract owner through several onlyOwner functions. This centralization poses a risk as it places considerable trust and control in a single entity. If the owner's private key is compromised, it could lead to significant disruptions or misuse of the contract.

OnlyOwner Functions:

`manageOperatorPrivileges()` Owner can add or remove operators.
`updateTokenWhitelist()` Owner can add or remove tokens from the whitelist.
`changePredicateAddress()` Owner can change the predicate address.
`changeFeeCollectorAddress()` Owner can change the fee collecter address.
`setMinimumFee()` Owner can set the minumum fee.
`withdraw()` Owner can withdraw leftover tokens.

Recommendation:
Use a multi-signature wallet for executing onlyOwner functions. This requires multiple authorized signatures to approve critical actions, reducing the risk of a single point of failure.

Client comment: Ownership will be transferred to a multisig after deployment.

Missing Handling for Tokens with Transfer Fees in fillOrders()

Location: AdvancedOrderEngine.sol
The fillOrders() function in AdvancedOrderEngine.sol does not account for tokens that force a fee on transfers, such as USDT. This results in a discrepancy in token balances after order fulfillment, breaking the invariant that the vault must not retain nor leak funds post-fulfillment of orders:
`The vault must not keep funds after the fullfilment of orders. I.e balance[before] == balance[after].`

Recommendation:

Updated the fulfillOrders() function to correctly handle tokens with transfer fees. Ensure that the protocol accounts for fees when calculating token balances post-fulfillment of orders to avoid the vault leaking funds. Carefully picking tokens that do not impost fee-on-transfer is a solution to be considered.

Client comment:
Defense Reason: Design Choice. Defense:
Whitelisted Tokens Only:

- Our platform exclusively supports whitelisted tokens, ensuring that only pre-approved tokens are used within the system. This significantly reduces the risk of encountering tokens with unexpected transfer fees.

Specific Issue with USDT:

- The primary concern revolves around USDT, as it is currently the main token with a transfer fee. We acknowledge this limitation and have plans in place to address it when necessary.

Future Adaptability:

- When USDT or any other token introduces a transfer fee, we have the flexibility to deploy a new contract. This is feasible because our current contract does not store significant data; it primarily functions as a gateway to validate orders before execution.

Simplification and Complexity Management:

- Handling tokens with transfer fees introduces additional complexity into the contract logic. To maintain the system's simplicity and reliability, we have decided not to address this issue at the present moment. This decision allows us to focus on core functionalities and ensure robust performance for the majority of use cases.

MEDIUM-3 | ACKNOWLEDGED

Orders Might Revert Since Fee Is Optional

feeAmounts on orders are supposed to be optional (L12 OrderEngine.sol) meaning orders can have `0` as feeAmounts too.
When an order is processed (in full or partial) the function _validateFee is used to validate the fee with respect to the minimum fee , but if an order has `0` as feeAmounts the call to _validateFee would revert (this is because executedFeeAmount would be calculated as `0` at L611 and the condition at L654 would revert) . Therefore , it would be impossible to process such an order.

Recommendation:
Only call _validateFee if order.feeAmounts > `0`

Client comment: Defense: When we say that fee is optional it means that protocol owners have the option to forgo the fee or to set one the check `_validateFee if order.feeAmounts >.Auditor's acceptable findings by design choice. `0` is important because we want to ensure that the protocol's demanded fee is being paid.

MEDIUM-4 | ACKNOWLEDGED

filledSellAmount Would Be Incorrect For Fee-On-Transfer Tokens

When an order is filled partially _processPartiallyFillableOrder() is invoked at L477 , and updated filledSellAmount by executedSellAmount at L608.
The assets are transferred at L527 (order maker transfers) where the amount is executedSellAmount but since the token is a fee-on-transfer token the actual sent amount would be less than executedSellAmount and the update at L477 would be incorrect.

Recommendation:
Handle fee-on-transfer tokens carefully or have a blacklist for such tokens.

Client comment: For now we are not handling this case, because it adds an extra layer of complexity. Once usdt enables fee-on-transfer, we will deploy v2 if needed. For other tokens, they will simply not be whitelisted.

MEDIUM-5 | ACKNOWLEDGED

getUsdPrice WILL NOT WORK WITH COMPLEX/LARGE PATHS

The intended behavior of the `getUsdPrice()` function in UsdOracle.sol is to return the amount in USDT, given a token, a token amount, and a period.

The current implementation of `getUsdPrice()` will only work if there is either a direct pool of the given token and USDT, or if there is an available path in the form of token-WETH-USDT. This means that if the given token does not have a direct pool with WETH or USDT, then it will not be possible to calculate a reliable USDT amount

Recommendation:
If other paths are to be considered for USD conversion calculation, then the implementation should be changed to allow for such behavior.

Client comment: This helper contract is used on the backend for matchmaking purposes. Since a token must have a pair with either WETH or USDT to exist on ElektrikDex, our design ensures reliable and consistent price calculations. On our frontend, we will only use whitelisted tokens that meet this criterion. This assumption and requirement have now been clearly documented within the contract to prevent any ambiguity. This approach simplifies the system and aligns with our use case, which does not require handling additional token paths for USD conversion.

MEDIUM-6 | ACKNOWLEDGED

TWAP CAN BE MANIPULATED IF THE PERIOD IS SHORT ENOUGH

The function `getUsdPrice` in `UsdOracle.sol` is used for converting an amount of token to usd. This function gets `_period` as input. `_period` is the number of seconds from which to calculate the TWAP. If `_period` is set to a short value then the price can be manipulated.

Recommendation:
Add a check that ensures that `_period` is at least `30` minutes to avoid price manipulation.

Client comment: Oracle contract is a helper contract, and we want it to be as flexible as possible. Also we are using the same contract at frontend for spot prices. Where twap manipulation is concerned the back-end will send the period as `30`.

LOW-1 | ACKNOWLEDGED

USE A MULTISIG ACCOUNT FOR THE OWNER

The functions `updateUsdt()`, `updateWeth()`, `updateOracle()` are onlyOwner callable. It means that the owner has the power of changing important variable values for the core behavior so it is recommendable to use a multisig account in order to avoid updating these variable by error or to avoid getting owner account compromised.

```
function updateUsdt(address _usdt) external onlyOwner {
        usdt = _usdt;
    }

    function updateWeth(address _weth) external onlyOwner {
        weth = _weth;
    }

    function updateOracle(address _oracle) external onlyOwner {
        oracle = IStaticOracle(_oracle);
    }
```

Recommendation:
Use a multi-sig wallet with a timelock as the owner.

Client comment: Ownership transfer after deployment.

LOW-2 | RESOLVED

LACK OF EVENTS

In the UsdOracle contract, the updateUsdt(), updateWeth(), and updateOracle() functions don’t emit events when key values are updated.

Recommendation:
Add events according to the states to be updated.

LOW-3 | RESOLVED

MISSING SANITY CHECKS FOR IMPORTANT VARIABLES

There are some important variables in the `UsdOracle.sol` contract that are not checked before being set in the constructor and also in the functions used for updating their values.

Recommendation:
Add a check to ensure that the set values for `usdt`, `weth` and `oracle` in the constructor and in `updateUsdt()`, `updateWeth()` and `updateOracle()` and not equal to address(0).

LOW-4 | ACKNOWLEDGED

Temporary DOS if one order contained a blacklisted address

The `fillOrders()` function in the `AdvancedOrderEngine` smart contract will revert the entire transaction if it encounters an order that contains a blacklisted address from a token like USDC. This behavior could disrupt the processing of valid orders and lead to inefficiencies, especially if the transaction includes multiple orders and only one is problematic.

Recommendation:
Consider skipping orders that have a blacklisted address and process the rest e.g. use continue rather than revert. or use an off-chain monitoring service to keep track of blacklisted addresses used by the protocol

Client comment: Backend will be responsible to first simulate and then submit the order, if a case arises, where a blacklisted address submits and order the backend will be responsible to choose a different candidate's order.

Owner can renounce ownership

The `Ownable2Step and Ownable` contracts includes a function named `renounceOwnership()` which can be used to remove the ownership of the contract.
If this function is called on the `AdvancedOrderEngine` contract, it will result in the contract becoming disowned. This would subsequently break several critical functions of the protocol that rely on `onlyOwner` modifier like:

```
manageOperatorPrivileges()
updateTokenWhitelist()
changePredicateAddress()
changeFeeCollectorAddress()
setMinimumFee()
withdraw()
```

Recommendation:
override the function to disable its functionality, ensuring the contract cannot be disowned e.g.

```
function renounceOwnership() public override onlyOwner {
revert ("renounceOwnership is disabled");
}
```

Improper Order of Modifiers in fillOrders()

The `fillOrders()` function of `AdvancedOrderEngine` contract currently places the `nonReentrant` modifier after the `onlyOperator` modifier. The order of modifiers can influence the behavior of a function, and it is generally recommended to place the `nonReentrant` modifier before other modifiers to ensure reentrancy protection is applied first.

File: AdvancedOrderEngine.sol

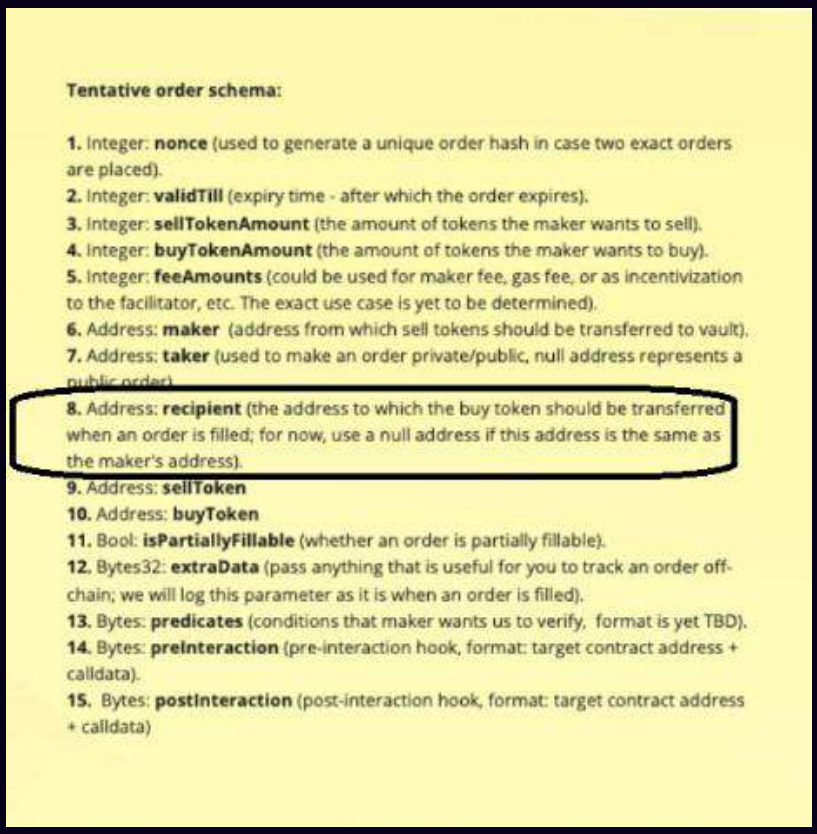
```
311:     function fillOrders(
..
319:     ) external onlyOperator nonReentrant {
```

Recommendation:

Reorder the modifiers so that `nonReentrant` is placed before `onlyOperator`. This ensures that reentrancy protection is applied as the first check.

Misleading Documentation for recipient Address Handling

The documentation for the `AdvancedOrderEngine` contract here specifies that if a user wants the recipient address to be the same as the maker address, they should set the recipient address to the null address `(address(0))`. However, the smart contract implementation reverts the transaction if the recipient address is the null address, leading to a discrepancy between the documentation and the actual behavior of the contract.



Tentative order schema:

1. Integer: **nonce** (used to generate a unique order hash in case two exact orders are placed).
2. Integer: **validTill** (expiry time - after which the order expires).
3. Integer: **sellTokenAmount** (the amount of tokens the maker wants to sell).
4. Integer: **buyTokenAmount** (the amount of tokens the maker wants to buy).
5. Integer: **feeAmounts** (could be used for maker fee, gas fee, or as incentivization to the facilitator, etc. The exact use case is yet to be determined).
6. Address: **maker** (address from which sell tokens should be transferred to vault).
7. Address: **taker** (used to make an order private/public, null address represents a public order).
8. Address: **recipient** (the address to which the buy token should be transferred when an order is filled; for now, use a null address if this address is the same as the maker's address).
9. Address: **sellToken**
10. Address: **buyToken**
11. Bool: **isPartiallyFillable** (whether an order is partially fillable).
12. Bytes32: **extraData** (pass anything that is useful for you to track an order off-chain; we will log this parameter as it is when an order is filled).
13. Bytes: **predicates** (conditions that maker wants us to verify, format is yet TBD).
14. Bytes: **preInteraction** (pre-interaction hook, format: target contract address + calldata).
15. Bytes: **postInteraction** (post-interaction hook, format: target contract address + calldata)

File: AdvancedOrderEngine.sol

```
533:     function _validateOrder(
534:         OrderEngine.Order calldata order,
535:         uint256 executedSellAmount,
536:         uint256 executedBuyAmount,
537:         bytes32 orderHash
538:     ) private view {
..
```

```
569:        // Revert if any address in the order is zero.
570:        if (
571:            order.maker == address(0) ||
572:            order.buyToken == address(0) ||
573:            order.sellToken == address(0) ||
574:            order.recipient == address(0)
575:        ) {
576:            revert ZeroAddress();
577:        }
```

Recommendation:
Align the contract behavior with the documentation or update the documentation to reflect the actual behavior of the contract.

LOW-8 | RESOLVED

Missing Sanity checks

It was found that the constructor of the `Airdrops.sol` smart contract does not include checks to validate the provided parameters are not zero. Although this is not critical, failing to perform this validation can lead to unexpected behaviors, specially because there are no setter functionalities in the contract to update the variables values.

File: Airdrops.sol
```
30:     constructor(bytes32 _merkleRoot, address _rewardContract) {
31:         merkleRoot = _merkleRoot;
32:         rewardContract = _rewardContract;
33:     }
```

Recommendation:
it is recommended to implement a check within the constructor to ensure that the provided parameters are not zero.

LOW-9 | RESOLVED

rewardContract and merkleRoot Cannot be updated

The `Airdrops.sol` smart contract does not provide setter functions for updating the `rewardContract` and `merkleRoot` after deployment. This design choice ensures immutability, but it also means that if an incorrect values are provided during the deployment of the contract to the constructor, these variables cannot be updated, potentially rendering the contract non-functional.

Recommendation:
Consider adding an `onlyOwner` protected setter functions to update the `rewardContract` and `merkleRoot` in case incorrect values are provided during deployment.

LOW-10 | RESOLVED

Incorrect Event Emission in Constructor

Location: AdvancedOrderEngine.sol

The `feeCollectorAddr` parameter is incorrectly emitted as part of the `PredicatesChanged` event in the constructor of the AdvancedOrderEngine contract, where it should be `predicatesAddr` instead.

```
emit PredicatesChanged(address(0), feeCollectorAddr);
```

Recommendation:
Update the event emission in the constructor to correctly emit `predicatesAddr` instead of `feeCollectorAddr` to accurately reflect the changes made.

LOW-11 | ACKNOWLEDGED

Missing Context Header in for merkle proof

Location: Airdrops.sol

The function `claim()` in Airdrops.sol does not include a header to provide context of the proof (e.g. address(this), chain id, version) which can lead to potential vulnerabilities.

Recommendation:
Include the necessary context header in the `claim()` function to ensure security and prevent potential vulnerabilities. This can help in verifying the legitimacy of the claim and enhance the overall security of the smart contract.

Client comment: Not applicable, since airdrop is only happening on one chain and for airdrops of new phases new contracts will be deployed. This disclaimer is updated in the documents and comments were updated.

LOW-12 | RESOLVED

Unused Return Value from mint() Function

Location: Airdrops.sol

In the `claim()` function of Airdrops.sol, the return value from the `mint()` function is not being used or captured for processing, potentially leading to unintended consequences or overlooked error handling.

Recommendation:
Ensure to capture and handle the return value from the `mint()` function appropriately, such as checking for success or failure status and reacting accordingly based on the result to maintain the integrity and correctness of the operation.

Commented-Out Code Results in Unnecessary Event Emission for Unchanged Whitelist Status

The `AdvancedOrderEngine` contract includes commented-out code that originally performed a check for unchanged access status of whitelisted tokens. This check was designed to revert the transaction with an `AccessStatusUnchanged()` error if the access status remained the same. However, the code is currently commented out, meaning this check is not executed. This might lead to confusion as the contract would emit an event even if there were no changes made.

File: AdvancedOrderEngine.sol

```
162:     function updateTokenWhitelist(
163:         address[] calldata tokens,
164:         bool[] calldata access
165:     ) external onlyOwner {
..
187:             // // Revert if the access status remains unchanged.
188:             // if (isWhitelistedToken[tokens[i]] == access[i])
189:             //     revert AccessStatusUnchanged();
190:             // }
191:
192:             isWhitelistedToken[tokens[i]] = access[i];
193:
194:             emit WhitelistStatusUpdated(tokens[i], access[i]);
```

Recommendation:
Consider Uncommenting the code to avoid unchanged status being emitted again.

Uninitialized minimumFee and Ability to Set minimumFee to Zero

The `minimumFee` variable is not initialized in the constructor, which means it defaults to zero. Additionally, the `setMinimumFee()` function allows the `minimumFee` to be explicitly set to zero. This might be acceptable depending on the intended functionality of the contract, but it is important to confirm whether this behavior aligns with the desired protocol operations.

File: AdvancedOrderEngine.sol

```
262:     function setMinimumFee(
263:         uint _fee
264:     ) external onlyOwner {
265:         if(_fee > 2000) revert ExceedsTwentyPercent();
266:         // Local copy to save gas.
267:         uint256 currentMinFee = minimumFee;
268:
269:         emit MinimumFeeChanged(currentMinFee, _fee);
270:
271:         minimumFee = _fee;
272:     }
```

Recommendation:
Review the intended design and functionality of the protocol to determine whether minimumFee being zero is acceptable.

Client comment: it was a design choice

INFORMATIONAL-3 | RESOLVED

Floating Pragma and Outdated Solidity version

The `Airdrops.sol and ClaimLib.sol` smart contracts uses a floating pragma version `(^0.8.13)`. Contracts should be deployed using the same compiler version and settings as were used during development and testing. Locking the pragma version helps ensure that contracts are not inadvertently deployed with a different compiler version. Using an outdated pragma version could introduce bugs that negatively affect the contract system, while newly released versions may have undiscovered security vulnerabilities.
Additionaly, There are known issues that affects 0.8.13, read more about the issues here.

Recommendation:
Consider locking the pragma version to a specific, tested version to ensure consistent compilation and behavior of the smart contract.

INFORMATIONAL-4 | RESOLVED

Public Function Could Be Marked External

The `claim()` function of the `Airdrops.sol` contract is marked as public but it could be marked as external instead. While both public and external functions can be called from outside the contract, marking a function as external is generally more gas-efficient when it is not intended to be called internally.

File: Airdrops.sol

```
38:    function claim(bytes32[] memory merkleProof, uint amount) public {
  ..
51:    }
```

Recommendation:
Consider changing the visibility of the claim function from public to external to optimize gas usage.

Lack of input validation in or() and and() functions

Location: Predicates.sol
The functions `or()` and `and()` in the Predicates.sol contract do not validate the `offsets` input, which can lead to errors and reverts during execution. The code snippet provided shows that the `previous` and `current` variables derived from `offsets` need validation to ensure `previous` is less than `current`. This lack of input validation can result in ambiguous errors and difficulties in debugging.

```
uint256 previous;
 for (
     uint256 current;
     (current = uint32(offsets)) != 0;
     offsets >>= 32
 ) {
     (bool success, uint256 res) = staticcallForUint(
         address(this),
         data[previous:current]
     ...
```

Recommendation:
Add input validation checks in the `or()` and `and()` functions to ensure that the `offsets` parameter is correctly formatted and that `previous` is always less than `current`. This can help prevent unexpected reverts and improve the robustness of the contract. Implementing proper input validations will enhance the readability, security, and reliability of the contract.

Client comment: Forked from CoW swap, will remain the same.

INFORMATIONAL-6 | RESOLVED

POOL PAIR IS NOT CHECKED IF IT IS SUPPORTED

The `getUsdPrice()` function in the `UsdOracle.sol` contract checks if a pair is supported before querying all pools for that pair. To check if a pair is supported a call to `oracle.isPairSupported` is executed. It is checked for the case of `_baseToken/usdt` and `_baseToken/weth` but not for the case of `weth/usdt`. The pair WETH/USDT will normally be supported, as it is one of the main pairs in most chains. However, if the protocol is deployed to a new chain, it could happen that the pair is not supported yet.

Recommendation:
Add a `oracle.isPairSupported(weth, usdt);` call and a `require(result, "PathToUsdtPoolNotAvailable");` for checking the returned value before executing the call for querying all weth/usdt pools `oracle.getAllPoolsForPair(weth, usdt);`

INFORMATIONAL-7 | RESOLVED

getUsdPrice DOES NOT RETURN USD PRICE

The function `getUsdPrice()` in `UsdOracle.sol` contract return the amount of base token in terms of USDT. However, the name of the function and some dev comments like: `/// @notice converts token to usd amount` suggest that the function is returning the amount in USD instead of USDT. It is important to remark that USDT price may not correspond to USD. The function name `getUsdPrice()` can also confuse and can suggest that the return amount is the price of 1 USDT instead of the amount in USDT terms.

Recommendation:
Change the function name to `getPriceInUsdt` and adjust comments.

INFORMATIONAL-8 | RESOLVED

CATCH ARRAY LENGTH

Storing the array length in a variable before the loop starts reduces the number of read operations, resulting in a saving of approximately 3 gas units per iteration. This practice can yield substantial gas savings, particularly in situations involving loops that iterate over large arrays.

Static.sol implements several 'for loops' that can save gas if array.length is catched before the `for loop`. In the StaticOracle contract, the addNewFeeTier function reads _knownFeeTiers.length at each iteration of the for loop.

Recommendation:
Catch array.length before the `for loop` to save gas.

INFORMATIONAL-9 | RESOLVED

USE ++i INSTEAD OF i++

The post-increment operation, i++ , requires more gas compared to pre-increment (++i). Post-increment involves both incrementing the variable i and returning its initial value, which necessitates the use of a temporary variable. This additional step results in extra gas consumption, approximately 5 gas units per iteration

StaticOracle.sol implements several for loops that can save gas if i is pre incremented.

Recommendation:
Use `++i` instead of `i++` in for loops.

	./Airdrops.sol ./libraries/ClaimLib.sol ./AdvancedOrderEngineErrors.sol ./Predicates.sol ./AdvancedOrderEngine.sol ./libraries/Decoder.sol ./libraries/OrderEngine.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./Vault.sol **./Helper/TargetContract.sol** **./Helper/VerifyPredicatesLogic.sol** **./Helper/GenerateCalldata.solLink** **./UsdOracle.sol** **./StaticOracle.sol** **./libraries/OracleLibraryPlus.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Elektrik in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the Elektrik contract/s requirements for details about issuance amounts and how the system handles these.

OracleLibraryPlusMock
- ✓ should calculate the arithmetic mean tick and harmonic mean liquidity correctly (72ms)
- ✓ should round the arithmetic mean tick to negative infinity

StaticOracle

supportedFeeTiers
- ✓ should return the default fee tiers (39ms)
- ✓ should revert when adding an already supported fee tier (38ms)
- ✓ should revert when adding an invalid fee tier

isPairSupported

when the given pair has no pools
- ✓ it is not supported (51ms)

when the given pair has at least one pool
- ✓ it is supported
- ✓ it is also supported in reverse order

_getPoolsForTiers

when sending no fee tiers
- ✓ returns empty array

when sending fee tiers but none have pool
- ✓ returns empty array

when sending fee tiers and some have pools
- ✓ returns the ones that have pools (43ms)

getAllPoolsForPair

when there are no pools
- ✓ an empty array is returned

when there are some pools
- ✓ they are returned

quoteAllAvailablePoolsWithTimePeriod

when there is a single pool

and it doesnt have an observation oldest than the period
- ✓ tx gets reverted with 'No defined pools' error (44ms)

and it has an observation oldest than the period
✓ queries the pool
when there are multiple pools
and none have an observation oldest than the period
✓ tx gets reverted with 'No defined pools' error
and some have an observation oldest than the period
✓ queries the correct pools (46ms)
quoteAllAvailablePoolsWithOffsettedTimePeriod
when there is a single pool
and it doesnt have an observation oldest than the period
✓ tx gets reverted with 'No defined pools' error
and it has an observation oldest than the period
✓ queries the pool
when there are multiple pools
and none have an observation oldest than the period
✓ tx gets reverted with 'No defined pools' error
and some have an observation oldest than the period
✓ queries the correct pools (43ms)
quoteSpecificFeeTiersWithTimePeriod
when quoting fee tiers that do not have pools
✓ tx gets reverted with 'Given tier does not have pool' error
when quoting fee tiers that have pools but do not support period
✓ tx gets reverted with 'Given tier does not have pool' error
when quoting fee tiers that do have pools that support period
✓ correct pools get queried (50ms)
quoteSpecificFeeTiersWithOfsettedTimePeriod
when quoting fee tiers that do not have pools
✓ tx gets reverted with 'Given tier does not have pool' error
when quoting fee tiers that have pools but do not support period
✓ tx gets reverted with 'Given tier does not have pool' error
when quoting fee tiers that do have pools that support period
✓ correct pools get queried

StaticOracle

constructor
when contract is initiated
✓ factory is set
✓ cardinality per minute is set
✓ default fee tiers are set
supportedFeeTiers
when no added tiers
✓ returns default fee tiers
when tiers were added
✓ returns correct fee tiers

prepareAllAvailablePoolsWithTimePeriod
when called
✓ increases cardinality correctly for pools
prepareSpecificFeeTiersWithTimePeriod
when sending tiers that do not have pool
✓ tx reverts with message
when all sent tiers have pools
✓ increases cardinality correctly for pools
prepareSpecificPoolsWithTimePeriod
✓ increases cardinality correctly for pools
prepareAllAvailablePoolsWithCardinality
when called
✓ increases cardinality correctly for pools
prepareSpecificFeeTiersWithCardinality
when sending tiers that do not have pool
✓ tx reverts with message
when all sent tiers have pools
✓ increases cardinality correctly for pools
prepareSpecificPoolsWithCardinality
✓ increases cardinality correctly for pools
addNewFeeTier
when trying to add a non-factory fee tier
✓ tx gets reverted with 'Invalid fee tier' message
when adding an already added fee tier
✓ tx gets reverted with 'Tier already supported' message
when adding valid fee tier
✓ gets added to known tiers
_getCardinalityForTimePeriod
✓ calculates cardinality for time period correctly
_prepare
✓ increases cardinality correctly for pools
_quote
when not quoting any pool
✓ tx reverts with message
when quoting an offsetted spot price
✓ tx reverts with message
_getQueryablePoolsForTiers
when period is zero
✓ returns all pools for that tier
when period is not zero
and all pool's observations are bigger or equal than period
✓ returns all pools

and not all pool's observations are bigger or equal than period
✓ returns only those who are

UsdOracle

getUsdPrice

✓ should return the base amount if the base token is USDT
✓ should return the price in USDT if the pair is supported
✓ should return the price in USDT via WETH if the direct pair is not supported
✓ should revert if no path to USDT is available

update functions

✓ should update USDT address only by owner (158ms)
✓ should update WETH address only by owner (87ms)
✓ should update Oracle address only by owner (48ms)

Ran 2 tests for test/Airdrops.t.sol:AirdropTest
[PASS] testClaim() (gas: 175338)
[PASS] testConstructorSetsMerkleRootAndRewardContract() (gas: 545130)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 12.80ms (5.84ms CPU time)

Ran 9 tests for test/unit/fuzz/PredicatesLogic.t.sol:VerifyPredicatesLogicTest
[PASS] test_and_lt_gt(uint256,uint256,uint256,uint256) (runs: 261, μ: 46955, ~: 46576)
[PASS] test_eq(uint256,uint256) (runs: 261, μ: 31449, ~: 31449)
[PASS] test_eq_edgeCases() (gas: 52038)
[PASS] test_gt(uint256,uint256) (runs: 261, μ: 31145, ~: 31321)
[PASS] test_gt_edgeCases() (gas: 59029)
[PASS] test_lt(uint256,uint256) (runs: 261, μ: 31339, ~: 31163)
[PASS] test_lt_edgeCases() (gas: 59351)
[PASS] test_not() (gas: 26989)
[PASS] test_or_lt_gt(uint256,uint256,uint256,uint256) (runs: 261, μ: 45280, ~: 45693)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 76.45ms (225.07ms CPU time)

Ran 100 tests for test/test.t.sol:Elektrik_V2_Tests
[PASS] testAsymetricFillOrKillOrders() (gas: 375290)
[PASS] testAsymetricPartiallyFillOrders() (gas: 358021)
[PASS] testCancelOrderAccessDenied() (gas: 21040)
[PASS] testCancelOrderAlreadyFilled() (gas: 63579)
[PASS] testCancelOrderSuccess() (gas: 63674)
[PASS] testChangeFeeCollectorAddressOnlyOwner() (gas: 12060)
[PASS] testChangeFeeCollectorAddressSameAddressRevert() (gas: 17910)
[PASS] testChangeFeeCollectorAddressSuccess() (gas: 22429)
[PASS] testChangeFeeCollectorAddressZeroAddressRevert() (gas: 15785)
[PASS] testChangePredicateAddressOnlyOwner() (gas: 11907)
[PASS] testChangePredicateAddressSameAddressRevert() (gas: 17356)

[PASS] testChangePredicateAddressSuccess() (gas: 21760)
[PASS] testChangePredicateAddressZeroAddressRevert() (gas: 15866)
[PASS] testCheckPredicateFailure() (gas: 17382)
[PASS] testCheckPredicateStaticCallFailure() (gas: 18211)
[PASS] testCheckPredicateSuccess() (gas: 17082)
[PASS] testConstructorInitialization() (gas: 15558)
[PASS] testConstructorRevertsOnZeroAddress() (gas: 176017)
[PASS] testCorrectDomainSeparator() (gas: 8101)
[PASS] testDomainSeparatorChangesWithChainID() (gas: 11636)
[PASS] testDomainSeparatorChangesWithContractAddress() (gas: 2460357)
[PASS] testDomainSeparatorConsistency() (gas: 7483)
[PASS] testDummyBool() (gas: 7304)
[PASS] testDummyFn() (gas: 7652)
[PASS] testDummyFn1() (gas: 6824)
[PASS] testDummyFn2() (gas: 6424)
[PASS] testDummyFn3() (gas: 7283)
[PASS] testExecutePreInteractionWithInvalidTarget() (gas: 211928)
[PASS] testExecutePreInteractionWithValidPreInteraction() (gas: 354275)
[PASS] testFacilatatorBorrowedAmounts() (gas: 406160)
[PASS] testFacilitatorSwap() (gas: 883067)
[PASS] testFacilitatorSwapExactInput() (gas: 1028934)
[PASS] testFillOrKillFee(uint256) (runs: 257, μ: 493296, ~: 502085)
[PASS] testFillOrderInChunks() (gas: 293173)
[PASS] testGenerateCalldata1() (gas: 16694)
[PASS] testGenerateCalldata2() (gas: 32289)
[PASS] testGenerateCalldataAnd_lt_gt() (gas: 37479)
[PASS] testGenerateCalldataNot() (gas: 15785)
[PASS] testGenerateCalldataOr_lt_gt() (gas: 37303)
[PASS] testGenerateCalldatadynamic_eq() (gas: 17605)
[PASS] testGenerateCalldatadynamic_gt() (gas: 17015)
[PASS] testGenerateCalldatadynamic_lt() (gas: 16728)
[PASS] testGetOrderHash() (gas: 32781)
[PASS] testGrantOperatorPrivileges() (gas: 42158)
[PASS] testOnlyOwnerCanManagePrivileges() (gas: 13127)
[PASS] testPartiallyFillFee(uint256) (runs: 257, μ: 440384, ~: 451201)
[PASS] testPartiallyFillFee2(uint256) (runs: 257, μ: 441399, ~: 452441)
[PASS] testPredicateANDFail1_LessThanAndGreaterThanConditions() (gas: 176892)
[PASS] testPredicateANDFail1_TwoGreaterThanConditions() (gas: 170853)
[PASS] testPredicateANDFail_LessThanAndGreaterThanConditions() (gas: 175682)
[PASS] testPredicateANDFail_TwoGreaterThanConditions() (gas: 172349)
[PASS] testPredicateFail_ArbitrarvStaticCall() (gas: 156444)

[PASS] testPredicateFail_EqualCondition() (gas: 156774)
[PASS] testPredicateFail_NotCondition() (gas: 156053)
[PASS] testPredicateOR_GreaterThanCondition() (gas: 331239)
[PASS] testPredicateOR_LessThanAndGreaterThanConditions() (gas: 319174)
[PASS] testPredicateOR_LessThanCondition() (gas: 331895)
[PASS] testPredicateOR_TwoLessThanConditions() (gas: 320233)
[PASS] testPredicate_LessThanCondition() (gas: 311941)
[PASS] testProcessPartiallyFillableOrderExceedsOrderSellAmount() (gas: 102267)
[PASS] testProcessPartiallyFillableOrderLimitPriceNotRespected() (gas: 81697)
[PASS] testProcessPartiallyFillableOrderSuccess() (gas: 204777)
[PASS] testRevokeOperatorPrivileges() (gas: 31530)
[PASS] testRingOrders() (gas: 402018)
[PASS] testSetMinimumFeeExceedsLimit() (gas: 13796)
[PASS] testSetMinimumFeeOnlyOwner() (gas: 11685)
[PASS] testSetMinimumFeeSuccess() (gas: 39434)
[PASS] testStress() (gas: 34107668)
[PASS] testUnchangedAccessStatusRevert() (gas: 42864)
[PASS] testUpdateTokenWhitelistArraysLengthMismatchRevert() (gas: 15974)
[PASS] testUpdateTokenWhitelistEmptyArrayRevert() (gas: 15000)
[PASS] testUpdateTokenWhitelistOnlyOwner() (gas: 16161)
[PASS] testUpdateTokenWhitelistSuccess() (gas: 46533)
[PASS] testUpdateTokenWhitelistZeroAddressRevert() (gas: 15125)
[PASS] testValidateInputArraysEmptyOrders() (gas: 61174)
[PASS] testValidateInputArraysLengthMismatchOrdersBuy() (gas: 69943)
[PASS] testValidateInputArraysLengthMismatchOrdersSell() (gas: 69361)
[PASS] testValidateInputArraysLengthMismatchOrdersSignatures() (gas: 75053)
[PASS] testValidateInputArraysSuccess() (gas: 246204)
[PASS] testValidateInteractionTargetInvalidSelfAddress() (gas: 159221)
[PASS] testValidateInteractionTargetInvalidZeroAddress() (gas: 158715)
[PASS] testValidateOrderAlreadyFilled() (gas: 265294)
[PASS] testValidateOrderBuyTokenNotWhitelisted() (gas: 76997)
[PASS] testValidateOrderExpired() (gas: 82529)
[PASS] testValidateOrderPrivateOrder() (gas: 77524)
[PASS] testValidateOrderSameBuyAndSellToken() (gas: 75689)
[PASS] testValidateOrderSellTokenNotWhitelisted() (gas: 78193)
[PASS] testValidateOrderSignatureContractFailure() (gas: 316977)
[PASS] testValidateOrderSignatureEOAFailure() (gas: 113728)
[PASS] testValidateOrderSignatureEOASuccess() (gas: 246604)
[PASS] testValidateOrderZeroBuyTokenAmount() (gas: 73288)
[PASS] testValidateOrderZeroExecutedBuyAmount() (gas: 72993)
[PASS] testValidateOrderZeroExecutedSellAmount() (gas: 72599)

```
[PASS] testValidateOrderZeroMakerAddress() (gas: 77828)
[PASS] testValidateOrderZeroRecipientAddress() (gas: 77796)
[PASS] testValidateOrderZeroSellTokenAmount() (gas: 73364)
[PASS] testWithdrawAccessControlAfterFillingOrders() (gas: 361006)
[PASS] testWithdrawRevertZeroAddressAfterFillingOrders() (gas: 366750)
[PASS] testWithdrawSuccessAfterFillingOrders() (gas: 357360)
[PASS] testZeroAddressRevert() (gas: 14351)
Suite result: ok. 100 passed; 0 failed; 0 skipped; finished in 1.94s (4.58s CPU time)

Ran 2 test suites in 1.95s (2.02s CPU time): 109 tests passed, 0 failed, 0 skipped (109 total tests)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
./Airdrops.sol	100%	100%	100%	100%
./libraries/ClaimLib.sol	100%	100%	100%	100%
./AdvancedOrderEngineErrors.sol	100%	100%	100%	100%
./Predicates.sol	91.67%	88.89%	100%	90%
./AdvancedOrderEngine.sol	91.15%	95.7%	95.65%	91.55%
./libraries/Decoder.sol	50%	50%	100%	100%
./libraries/OrderEngine.sol	100%	100%	100%	100%
/Vault.sol	100%	100%	100%	100%
./Helper/TargetContract.sol	100%	100%	100%	100%
./Helper/VerifyPredicatesLogic.sol	100%	100%	66.67%	100%
./Helper/GenerateCalldata.sol	98.59%	75%	100%	98.57%
./UsdOracle.sol	95%	83.33%	80%	94.73%
./StaticOracle.sol	95.38%	77.27%	86.95%	95.31%
./libraries/OracleLibraryPlus.sol	100%	100%	100%	100%
All files	**94.95%**	**90.41%**	**92.94%**	**95.37%**

We are grateful for the opportunity to work with the Elektrik team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Elektrik team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo