SMART CONTRACTS REVIEW

zokyo

March 18th 2024 | v. 1.0

Security Audit Score

PASS

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

ZOKYO AUDIT SCORING ZKDX

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.

HYPOTHETICAL 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
- 0 High issues: 0 points deducted
- 7 Medium issues: 4 resolved and 3 acknowledged issues = - 9 points deducted
- 12 Low issues: 4 acknowledged and 8 resolved issues = - 5 points deducted
- 1 Informational issue: 1 acknowledged issue = 0 points deducted

Thus, 100 - 9 - 5 = 86

TECHNICAL SUMMARY

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

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

Contract Status

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

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 zkDX 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 zkDX repository:
Repo: https://github.com/zkDX-DeFi/Smart_Contracts/tree/master/contracts/core

Last commit: 45ddd19ad0a30ab71880e3c442211cfdf63e0506

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

- contracts/core/PositionManager.sol
- contracts/core/BasePriceConsumer.sol
- contracts/core/Vault.sol
- contracts/core/VaultPriceFeed.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 zkDX 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. 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.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 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.

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 zkDX team and the zkDX 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	Non-concurrent buying and selling in Vault can lead to loss of funds	Medium	Acknowledged
2	Inadequate Validation in clearTokenConfig	Medium	Resolved
3	“isLeverageEnabled” is true by default	Medium	Resolved
4	Unbounded Iteration Leading to Potential Service Blockage	Medium	Resolved
5	Incorrect Fee Calculation Due to Outdated ZKUSDAmounts Value	Medium	Acknowledged
6	Preventing Unauthorized ZKLP Token Creation	Medium	Resolved
7	Governance control is centralized through a single gov address	Medium	Acknowledged
8	Use of single step ownership transfer	Low	Resolved
9	Lack of Event Emission After Sensitive Actions	Low	Resolved
10	Inadequate Handling of Fee on Transfer Tokens	Low	Acknowledged
11	Stale Price Exploitation Due to Manipulable validTime	Low	Resolved
12	Missing events for VaultPriceFeed	Low	Resolved
13	Usage of getPriceUnsafe should be avoided	Low	Resolved
14	Missing zero address check in setGov() of VaultPriceFeed	Low	Resolved
15	Use of pragma experimental ABIEncoderV2	Low	Acknowledged
16	Old Solidity Version	Low	Acknowledged

#	Title	Risk	Status
17	_liquidationFeeUsd and _fundingRateFactor Initialization Checks	Low	Resolved
18	Receiver Address Checks for Zero Address	Low	Resolved
19	Hardcoded Stable Token Price in VaultPriceFeed.sol	Low	Acknowledged
20	Avoid using TWAPs as PriceFeeds	Informational	Acknowledged

Non-concurrent buying and selling in Vault can lead to loss of funds

Description :
If the user deposits tokens to the contract directly and calls buyZKUSD() function, then this function can be frontrunned by users and attackers before other users, resulting in zkusd being minted to the attacker. This is because the _transferIn() function on line: 61 assumes that the user has already transferred the underlying token to the vault in the previous transaction, and _transferIn() just calculates the difference in the balances of Vault in terms of the token. For example, Alice deposits 100 RETH tokens to the contract. Now when Alice sends a transaction to call the buyZKUSD() function after sending tokens to the Vault, Eve frontruns her transaction leading to the ZKUSD tokens to Eve instead of Alice.

This could also result in loss of funds for the first user, if two users simultaneaously transfer their funds, but the second user calls buyUSD() instead of the first. Leading to all ZKUSD being minted to the second user.

Similar issue exists for sellZKUSD() function of the Vault as _transferIn() is used again for this function on line: 88.

Another attack scenario which exists here is when two users say Alice and Bob transer 10 ZKUSD and 1000 ZKUSD respectively to the Vault. Now if Alice calls the sellZKUSD() function before Bob, it is possible for her to sell Bob's ZKUSD and burn them too! Thus Alice would get the underlying tokens by burning ZKUSD resulting her getting the underlying tokens too which were actually meant for Bob to redeem.
This issue also exists in swap() as it is _transferIn() is being called in swap() too on line: 119.

Although this vulnerability is mitigated by higher-level contracts such as Router.sol, which performs token transfers and vault actions atomically, direct interactions with the Vault.sol contract by users bypassing these safeguards could still be susceptible.

Recommendation:
It is advised to use a safetransferFrom() so that users transfer the tokens to the contract in the same transaction as the buyZKUSD() (and _transferIn) to avoid the above attack scenarios.

Comments: The client has used a stricter onlyManager modifier that allows only accounts with manager role to call this function. This does reduce the risk. But again the modifier acts as a whitelist due to usage of isManager mapping which does that erase the risk fully.

MEDIUM-2 | RESOLVED

Inadequate Validation in clearTokenConfig

Description :
This function allows the removal of a token's configuration without ensuring there are no active or pending operations involving the token. If the token is in use (e.g., in open positions or liquidity pools), removing its configuration could lead to operations that are unable to be completed successfully, potentially resulting in locked funds or incorrect system behavior.

Recommendation:
Before allowing a token's configuration to be cleared, implement checks to ensure that the token is not currently being used in any active operations. Additionally, consider providing a grace period during which users can close positions or withdraw funds.

“isLeverageEnabled” is true by default

Description :
In the Vault.sol contract, there's a boolean called isLeverageEnabled that should be (false) to start. This ensures only PositionManager.sol can manage positions. But, the boolean is mistakenly left (true) when the contract is first set up. This mistake lets anyone directly use the Vault.sol to open positions right away, skipping important checks in _validateIncreaseOrder (PositionManager.sol).

https://github.com/zkDX-DeFi/Smart_Contracts/blob/35f1d4b887bd5b0fc580b7d9fe951c4b550c9897/contracts/core/Vault.sol#L148

Recommendation:
it is recommended to explicitly initialize the isLeverageEnabled flag to false within the Vault contract's constructor. This change ensures that leverage-related operations are disabled by default, requiring explicit authorization (typically from an admin or through a governance process) to enable such features

```sol
constructor() public {
    gov = msg.sender;
    isLeverageEnabled = false; // Ensure leverage is disabled by default.
}
```

Unbounded Iteration Leading to Potential Service Blockage

Description :
Vault.sol:setTokenConfig due to the absence of a limit on the number of whitelisted tokens. This can lead to scenarios where operations iterating over these tokens consume more gas than the maximum allowed in a block, potentially causing transactions to fail and, in severe cases, blocking the service entirely.

Recommendation:
Set a Maximum Cap: Implement a cap for the maximum number of whitelisted tokens. This ensures that the list cannot grow indefinitely and operations on the list stay within the block gas limit.

uint256 public constant MAX_WHITELISTED_TOKENS = 100; // Example maximum number

Incorrect Fee Calculation Due to Outdated ZKUSDAmounts Value

Description :
The ZKLP is engineered to rebalance using a dynamic fee structure (source: https://docs.zkdx.io/litepaper/usdzklp#usdzklp-rebalance) . However, in the VaultUtils#getFeeBasisPoints function, where the determination of whether an action is contributing towards or moving away from the desired rate relies on outdated values from the zkusdAmounts[_token] mapping.

This mapping holds the USD value cached at the time of an action's execution (https://github.com/zkDX-DeFi/Smart_Contracts/blob/35f1d4b887bd5b0fc580b7d9fe951c4b550c9897/contracts/core/VaultUtils.sol#L73). If there is a significant price fluctuation in the token value after this point, the cached value will not be updated to reflect this change. As a result, the rebalancing mechanism may inaccurately assess the direction of the action relative to the target weight, potentially leading to the imposition of incorrect fees. In the worst-case scenario, this could result in the rebalancing process moving in the opposite direction of its intended goal.

Exploit Scenario:
Assume the ZKLP aims for a 50-50 weight between DAI and USDT.
Initially, the pool contains 1 DAI (priced at $100) and $100 USDT, perfectly balanced.
If the price of DAI drops to $50, the pool would have $50 worth of DAI and $50 USDT, unbalancing the pool.
When a user attempts to buy ZKUSD with DAI at this point, the optimal behavior would be to encourage adding DAI to the pool by charging a lower fee. However, due to the outdated ZKUSD[DAI] = 100, the system incorrectly perceives adding DAI as diverging further from the target weight, thus charging an extra fee instead.

Recommendation:
Implementing real-time value updates or a mechanism that can more accurately determine the direction of balance relative to the target weight without relying solely on cached values could be potential solutions.

Preventing Unauthorized ZKLP Token Creation

Description :
The ZkdlpManager.sol:getAum function, presents two primary issues:

- Denial of Service (DoS) Risk: The function iterates over all tokens listed in the allWhitelistedTokens array. If the governance role, through VaultSettings.sol:setTokenConfig, adds a substantial number of tokens, it could lead to a situation where the loop exhausts the gas limit, causing a DoS scenario.
- Duplicate Token Counting: The Vault.sol:clearTokenConfig function fails to remove tokens from the allWhitelistedTokens array upon clearance. If a token is subsequently re-added, it results in its duplication within the array. Consequently, the getAum function would count such tokens multiple times, inaccurately increasing the Assets Under Management (AUM) calculation. This discrepancy adversely affects the ZkdlpManager.sol:_addLiquidity function, leading to the minting of fewer ZKLP tokens than appropriate (since an inflated AUM results in reduced mint amounts).

Exploit Scenario:
1. Governance adds the USDStable token to the pool.
2. USDStable was later removed due to declining reputation.
3. USDStable is re-added after its reputation and popularity recover.
4. ZKLP tokens are minted in smaller amounts due to the inflated AUM calculation.

Recommendation:
- Modify Vault.sol:clearTokenConfig to ensure that tokens are properly removed from the allWhitelistedTokens array when cleared. This change prevents the double-counting of tokens in AUM calculations.
- Consider implementing mechanisms to prevent potential DoS attacks due to excessive iteration. This could include setting a limit on the number of tokens that can be added.

```sol
function clearTokenConfig(address _token) external override {
    _onlyGov();
    require(whitelistedTokens[_token], "Token not whitelisted");
    totalTokenWeights = totalTokenWeights.sub(tokenWeights[_token]);
```

```
    // Properly removing the token from allWhitelistedTokens array
    for (uint i = 0; i < allWhitelistedTokens.length; i++) {
        if (allWhitelistedTokens[i] == _token) {
            allWhitelistedTokens[i] = allWhitelistedTokens[allWhitelistedTokens.length - 1];
            allWhitelistedTokens.pop();
            break;
        }
    }

   // Clearing token configurations
    delete whitelistedTokens[_token];
    delete tokenDecimals[_token];
    delete tokenWeights[_token];
    delete minProfitBasisPoints[_token];
    delete maxZkusdAmounts[_token];
    delete stableTokens[_token];
    delete shortableTokens[_token];
    whitelistedTokenCount = whitelistedTokenCount.sub(1);
}
```

Governance control is centralized through a single gov address.

Description :
The contract VaultPriceFeed employs a single gov address for critical governance actions, it introduces a significant risk by creating a single point of failure. If the private key associated with the gov address is compromised, an attacker gains the capability to manipulate crucial contract settings, potentially leading to system-wide vulnerabilities or exploitation.

https://github.com/zkDX-DeFi/Smart_Contracts/blob/35f1d4b887bd5b0fc580b7d9fe951c4b550c9897/contracts/core/VaultPriceFeed.sol#L13

Recommendation:
Decentralized Governance Control: Implement a multi-signature mechanism or a decentralized autonomous organization (DAO) structure for critical governance decisions.

Emergency Stop Mechanism: Introduce an emergency stop feature that can be activated by multiple parties in case of a detected compromise. This can halt critical contract functions until a thorough investigation is conducted.

Transparency and Monitoring: Maintain transparency about governance actions with the community or stakeholders. Implement monitoring and alert systems for governance-related activities to detect unusual patterns that may indicate a compromise.

LOW-1 | RESOLVED

Use of single step ownership transfer

Description :
In the Vault contract which allows changing the gov address in the abstract vaultSetting & In VaultPriceFeed it allowed to change gov address using setGov function However, this contract does not implement a 2-step-process for transferring ownership. If the admin's address is set incorrectly, this could potentially result in critical functionalities becoming locked.

Recommendation:
Consider implementing a two-step pattern. Utilize OpenZeppelin's Ownable2Step contract.

LOW-2 | RESOLVED

Lack of Event Emission After Sensitive Actions

Description :
The system exhibits a lack of event emissions following the execution of sensitive actions across multiple contracts, decreasing transparency and making tracking transactions difficult. Specifically:

- In the vault contract, the withdrawFees function allows the withdrawal of accumulated fees to any address but does not emit an event upon completion.
- The VaultPriceFeed contract interacts with the Pyth Oracle for managing price feeds of various tokens. It includes critical functions like setValidTime, setFeedIds, setPyth, and setStableToken that alter the state of the contract without emitting events after execution.

Recommendation:
It is recommended to define and emit events for all critical state-changing operations within the contracts, including successful fee withdrawals and any changes made through the VaultPriceFeed contract's functions. Details such as the token address, amount withdrawn, receiver's address, and the specific state changes made should be included in these events to enhance transparency and facilitate tracking.

Inadequate Handling of Fee on Transfer Tokens

Description :
The smart contract's mechanisms for handling token transfers, particularly in functions like directPoolDeposit, buyZKUSD, sellZKUSD, swap, increasePosition, decreasePosition, and related liquidity and position management operations, do not account for tokens that deduct fees on transfers. When interacting with fee on transfer (FoT) tokens, the actual amount received by the contract can be less than the amount sent, due to these tokens deducting a fee from transfers for rewards, burning, or other purposes. This discrepancy can lead to inconsistencies in accounting, where the contract believes more tokens are deposited or withdrawn than actually are, potentially affecting pool balances, user balances, and the accuracy of position valuations.

Recommendation:
Detect and Reject FoT Tokens: Implement checks to identify and reject fee on transfer tokens, preventing their use in the protocol. This can be done through static analysis or dynamic checks at the time of deposit or interaction.

Stale Price Exploitation Due to Manipulable validTime

Description :
The VaultPriceFeed contract fetches asset prices from the Pyth Oracle for financial operations within a DeFi platform. A critical vulnerability exists due to the mutable validTime state variable, which governs the maximum allowed age for price data. If validTime is set too high (either maliciously by a compromised governance or accidentally), the contract may accept stale prices as current, leading to potential exploitation in trading, loan issuance, and liquidation processes.

Scenario :
An attacker gains control over the governance address or persuades governance to increase the validTime significantly (e.g., to several days or more). They then wait for a significant price movement in an asset that would normally invalidate outdated oracle data. Because of the extended validTime, the contract continues to use the stale price, allowing the attacker to execute trades or loans against the DeFi platform at outdated and favorable rates, leading to potential profit at the expense of the platform and its users.

1. Initial Setup: The VaultPriceFeed contract has validTime set to 3 seconds, ensuring price data is fresh.
2. Compromise: The attacker, having governance control, sets validTime to `86400` seconds (1 day).
3. Exploitation: The attacker observes that the price of TokenA has significantly dropped in external markets but is still high in the Pyth Oracle due to a lack of updates within the last day. They use this stale price to liquidate positions or take out loans at far more favorable terms than the current market conditions would allow.

Recommendation:
it is recommended to implement a hard cap on the validTime variable. This hard cap would act as an upper limit to how long price data can be considered valid, regardless of governance actions

LOW-5 | RESOLVED

Missing events for VaultPriceFeed

Description :
Missing events for critical functions in VaultPriceFeed.sol such as setGov(), setValidTime(), setStableToken(), setPyth() and updatePriceFeeds().

Recommendation:
Consider adding events for functions with critical state changes as it would be a best practice for offchain monitoring.

LOW-6 | RESOLVED

Usage of getPriceUnsafe should be avoided

Description :
It is advised to use pyth.getPriceNoOlderThan() instead of pyth.getPriceUnsafe() in VaultPriceFeed.sol. This is because getPriceNoOlderThan() already has necessary sanity checks built in to ensure that the price is no older than the age time period. Also read the documentation of PythStructs.Price to understand how to use this safely.
Refer- https://github.com/pyth-network/pyth-sdk-solidity/blob/main/IPyth.sol#L36

Recommendation:
Use pyth.getPriceNoOlderThan() rather than using pyth.getPriceUnsafe().

LOW-7 | RESOLVED

Missing zero address check in setGov() of VaultPriceFeed

Description :
VaultPriceFeed.sol has missing zero address check in setGov() for _gov address parameter. This can lead to the gov being accidentally set as zero address, leading to onlyGov function being uncallable forever.
The same issue exists in the VaultSettings contract which the Vault.sol contract inherits. Setting gov as zero address here would lead to withdrawFees() being uncallable amongst other functions, leading to stuck fees in the Vault.

Recommendation:
It is advised to add a zero address check for the same.

LOW-8 | ACKNOWLEDGED

Use of pragma experimental ABIEncoderV2

Description :
The code contains "pragma experimental ABIEncoderV2;"
In the later Solidity versions it is no longer necessary to use the "experimental" version. Using experimental constructions is not recommended for production code.

See: Layout of a Solidity Source File — Solidity 0.8.25 documentation (soliditylang.org)

https://github.com/zkDX-DeFi/Smart_Contracts/blob/35f1d4b887bd5b0fc580b7d9fe951c4b550c9897/contracts/core/PositionManager.sol#L3

Recommendation:
Replace “pragma experimental ABIEncoderV2;” with “pragma abicoder v2”;

And make sure you use at least solidity version 0.7.5

LOW-9 | ACKNOWLEDGED

Old Solidity Version

Description :
Contracts Vault.sol, BasePriceConsumer.sol, PositionManager.sol are using solidity version 0.6.x, we recommend using the latest stable version of Solidity.

LOW-10 | RESOLVED

_liquidationFeeUsd and _fundingRateFactor Initialization Checks

Description :
The initialize function in Vault.sol should include checks to ensure that _liquidationFeeUsd and _fundingRateFactor are less than or equal to their respective maximum allowed values (MAX_LIQUIDATION_FEE_USD and MAX_FUNDING_RATE_FACTOR). This is important to prevent setting these parameters to values that could destabilize the contract's economic mechanisms or create unintended behavior.

Recommendation:
To address this, you can add `require` statements in the `initialize` function to validate these conditions:
require(_liquidationFeeUsd <= MAX_LIQUIDATION_FEE_USD, "Liquidation fee exceeds maximum");
require(_fundingRateFactor <= MAX_FUNDING_RATE_FACTOR, "Funding rate factor exceeds maximum");

LOW-11 | RESOLVED

Receiver Address Checks for Zero Address

Description :
Functions like sellZKUSD, buyZKUSD, swap, increasePosition, and decreasePosition should include checks to ensure that the _receiver address is not the zero address. This is a common practice to prevent funds from being sent to an unrecoverable address.

Recommendation:
require(_receiver != address(0), "Receiver cannot be the zero address");

LOW-12 | ACKNOWLEDGED

Hardcoded Stable Token Price in VaultPriceFeed.sol

Description :
In the VaultPriceFeed.sol contract, the price of stable tokens is hardcoded as 1e30, it assumes that all stable tokens will indefinitely maintain their peg to their underlying assets. This assumption poses a risk if a stable token significantly depegs, yet the system continues to treat it as if it were still pegged, potentially leading to inaccurate valuations and operations based on these prices.

https://github.com/zkDX-DeFi/Smart_Contracts/blob/35f1d4b887bd5b0fc580b7d9fe951c4b550c9897/contracts/core/VaultPriceFeed.sol#L27C1-L28C25

Recommendation:
Fallback Mechanism: In cases where a stable token's price deviates significantly from its peg, introduce fallback mechanisms such as temporary suspension of operations for the affected token or automatic adjustments to its valuation within the system.

Avoid using TWAPs as PriceFeeds

Description:
In the _update() function, the priceFeed is being assigned here:
IVaultPriceFeed priceFeed = IVaultPriceFeed(_IVault.priceFeed());

The price feed is assigned in the Vault contract. If TWAPs or Time Weighted Average Prices from DEXs are used for prices, then it can result it Oracle manipulation attacks.

Recommendation:
It is advised not to use any TWAP oracles as its price is manipulatable. It is advised to use oracles such as Chainlink, Tellor, Witnet, etc.

	contracts/core/PositionManager.sol contracts/core/BasePriceConsumer.sol contracts/core/Vault.sol contracts/core/VaultPriceFeed.sol
Reentrance	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

We are grateful for the opportunity to work with the zkDX 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 zkDX team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo