SMART CONTRACTS REVIEW

zokyo

March 28th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

ZOKYO AUDIT SCORING DEFACTOR

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:
- 2 Critical issues: 2 resolved = 0 points deducted
- 4 High issues: 4 resolved = 0 points deducted
- 8 Medium issues: 8 resolved = 0 points deducted
- 5 Low issues: 5 resolved = 0 points deducted
- 8 Informational issues: 4 resolved and 4 acknowledged = 0 points deducted

Thus, the score is 100

TECHNICAL SUMMARY

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

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

Contract Status

There were 2 critical issue found during the audit. (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 Defactor 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 5

Executive Summary 7

Structure and Organization of the Document 9

Complete Analysis 10

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the Defactor repository.
Repo: https://github.com/defactor-com/blockchain

Last commit - 3af91f92c34d67e3a8bc4129bb18e52965cd0a04

Contracts under the scope:

- ./erc-20-mixer/ERC20Mixer.interface.sol
- ./erc-20-mixer/ERC20MixerFactory.sol
- ./erc-20-mixer/ERC20Mixer.sol
- ./data-storage/DataStorage.sol
- ./data-storage/IDataStorage.sol
- ./pools/Pools.storage.sol
- ./pools/Pools.interface.sol
- ./pools/Pools.sol
- ./erc-20/ERC20Factory.sol
- ./erc-20/ERC20.sol
- ./vesting/ERC20LazyVesting.interface.sol
- ./vesting/ERC20LazyVesting.sol
- ./staking/Staking.interface.sol
- ./staking/Staking.storage.sol
- ./staking/Staking.sol
- ./erc-20-collateral-pool/ERC20CollateralPool.storage.sol
- ./erc-20-collateral-pool/AggregatorV3Interface.sol
- ./erc-20-collateral-pool/ERC20CollateralPool.interface.sol
- ./erc-20-collateral-pool/ERC20CollateralPool.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 Defactor 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 Testing contract/s logic against common and uncommon attack vectors.

Executive Summary

The Defactor protocol aims to innovate at the intersection of real world assets and decentralized finance. Defactor's FACTR token is a utility token that drives the technology's ecosystem. The revenue generated goes into the buyback contract supporting adoption. More usage of the ecosystem means more buybacks allowing growth to be aligned with token demand. In Defactor, there exists their core pillars including technology & innovation by pioneering innovation in RWAs and DeFI, governance allowing a collective group of individuals to collaborate and drive innovation in RWA tokenization, community to create a global support system and liquidity & markets which ensures the constant availability of market liquidity in addition to monitoring market conditions which is what this set of smart contracts addresses.

Zokyo was tasked with conducting the security reviews for the ERC20 collateral pools, the ERC20 mixer, the ERC20 token blueprint and factory, pools, staking, and vesting. The users can interact with the Defactor protocol in a multitude of ways whether it be for staking to stake various tokens and earn rewards, using the pools to allow users to create, manage and interact with funding pools using the most popular token standards such as ERC20, ERC721 or ERC1155, in addition to facilitating lending and borrowing positions through the ERC20 collateral pools.

Overall, the code is well thought out and well engineered in accordance with Defactor's guidelines and addresses a growing area in our industry however, additional natspec within the code itself could add to the code presentation and added context. The security team discovered issues ranging from Critical issues all the way down to information that aims to address best engineering practices. These issues revolved around root causes which stemmed from errors in calculations in addition to some logic errors, the way fees are handled and validation around return values from Chainlink oracles and reinforcement of best engineering practices such as Checks Effects and Interactions (CEI), and ensuring functions return key variables. Some issues were deemed to be within the protocol's risk appetite resulting in acknowledgement but still remained in this report to keep the users informed of the security review process.

After extensive discussions around the discovered issues, the protocol team promptly fixed the issues where a fix review took place to ensure that no additional bugs were introduced into the code base as a result of the fixes and to ensure that the discovered attack vectors were no longer exploitable. Whilst some unit testing does exist using the hardhat framework, there can always be more testing introduced to ensure the intended actions of the contracts in addition to fuzz testing to catch any edgecase scenarios. In addition to this, it can also be helpful to update the documentation on the Defactor development documents so that users can review the contracts intended purpose. As mentioned previously, natspec within the contracts themselves can also assist with this. Finally, a bug bounty program is also highly advised for the protocol's continuous commitment to ensuring security for the ecosystem and its users. We at Zokyo wish Defactor all the best of luck in their deployment to the relevant production blockchains!

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 Defactor team and the Defactor 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	The value of collateral amount in USDC is incorrectly calculated, leading to liquidations being more likely to like place and the amount of collateral returned being incorrect	Critical	Resolved
2	Incorrect fee is taken from the users due to hardcoded token decimals, leading to loss of funds for the protocol or the users	Critical	Resolved
3	Risk of Instant Liquidation and Front-Running After Protocol Resumption	High	Resolved
4	Chainlink Data Staleness Threshold is Too Wide	High	Resolved
5	Attack can steal funds from the Staking contract when certain staking/reward tokens are used	High	Resolved
7	Rewards Balance In The Staking Contract Is Not Considered Which May Result In Insolvency And Denial Of Withdrawals	High	Resolved
8	Withdrawal Function In Staking Allows Default Admins To Rug The Contract	Medium	Resolved
9	Collateral ERC20 Tokens With More Than 18 Decimals Are Incompatible In The ERC20CollateralPool	Medium	Resolved
10	Non implemented checks can lead to unexpected behaviors	Medium	Resolved
11	CEI pattern not followed allowing reentrancy attacks under certain conditions	Medium	Resolved
12	Sequencer Downtime Risk on L2 Chainlink Price Feeds	Medium	Resolved
13	The transfer of certain token's transfer can silently fail	Medium	Resolved

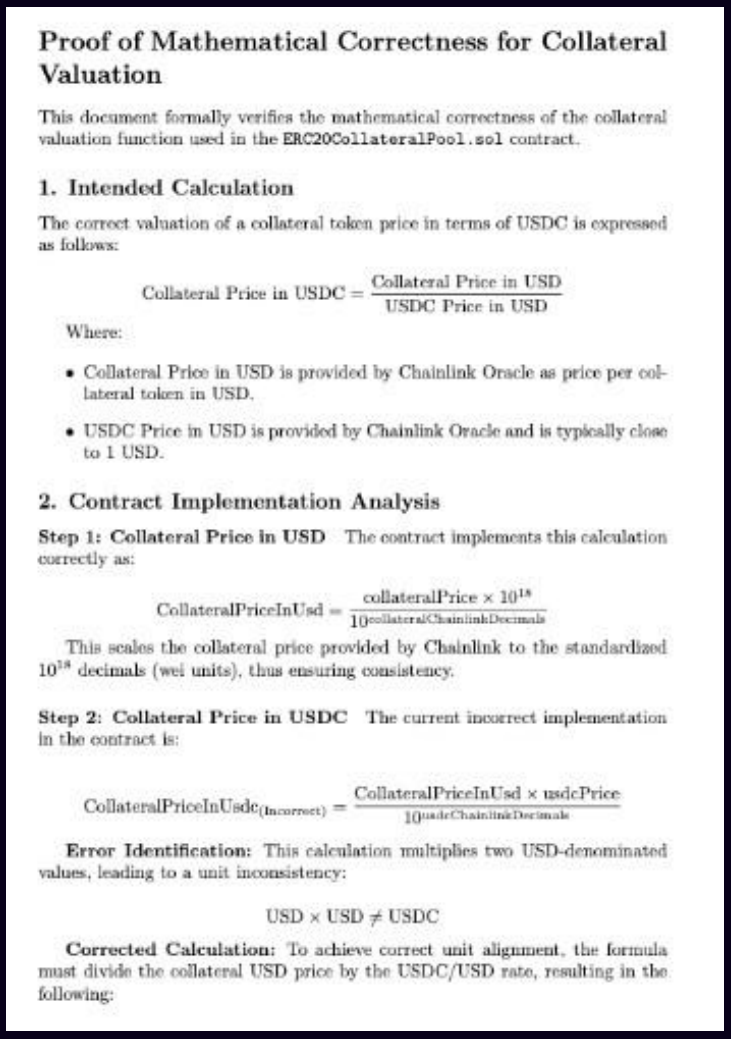
#	Title	Risk	Status
14	A pool using a free-on-transfer token as collateral will result in incorrect accounting of collateral amounts	Medium	Resolved
15	The amount of staked funds is not decreased when a user unstakes, leading to an incorrect track of funds	Medium	Resolved
16	Archiving A Pool In The Pools Contract Which Still Contains User Funds Will Result In Stuck Tokens	Low	Resolved
17	Collateral Token LTV Percentage Not Explicitly Validated Against Zero	Low	Resolved
18	Cross-Site Scripting (XSS) via Token Symbol Injection	Low	Resolved
19	Unchecked Array Length Leading to Excessive Gas Costs	Low	Resolved
20	ERC20CollateralPoolStorage initialization will revert due to token decimal not matching	Low	Resolved
21	Using deprecated `_setupRole()` function may lead to unexpected behaviors	Informational	Acknowledged
22	Some actions are centralized	Informational	Acknowledged
23	Variable with incorrect spelling	Informational	Resolved
24	It is an engineering best practice to use Openzeppelin libraries	Informational	Acknowledged
25	uint256 can never be lower than 0	Informational	Resolved
26	Hashes Can Be Used To More Efficiently Check If A Staking Plan Exists In The Staking Contract	Informational	Acknowledged
27	Unused Code In Staking.storage.sol	Informational	Resolved
28	Key Variables Are Not Returned In Certain Functions	Informational	Resolved

The value of collateral amount in USDC is incorrectly calculated, leading to liquidations being more likely to like place and the amount of collateral returned being incorrect.

Description:
The `ERC20CollateralPool.sol` smart contract implements a private `_estimateErc20Value()` function which is used to calculate the value of a pool's collateral amount in USDC. However, there is an error when calculating `collateralPriceInUsdc':

```
uint256 collateralPriceInUsdc =
            collateralPriceInUsd * uint256(usdcPrice) /
            (10 ** usdcChainlinkDecimals);
```

It can be observed that `collateralPriceInUsd` is multiplied to `usdcPrice` instead of divided,



Proof of Mathematical Correctness for Collateral Valuation

This document formally verifies the mathematical correctness of the collateral valuation function used in the `ERC20CollateralPool.sol` contract.

1. Intended Calculation

The correct valuation of a collateral token price in terms of USDC is expressed as follows:

$$\text{Collateral Price in USDC} = \frac{\text{Collateral Price in USD}}{\text{USDC Price in USD}}$$

Where:

- Collateral Price in USD is provided by Chainlink Oracle as price per collateral token in USD.
- USDC Price in USD is provided by Chainlink Oracle and is typically close to 1 USD.

2. Contract Implementation Analysis

Step 1: Collateral Price in USD The contract implements this calculation correctly as:

$$\text{CollateralPriceInUsd} = \frac{\text{collateralPrice} \times 10^{18}}{10^{\text{collateralChainlinkDecimals}}}$$

This scales the collateral price provided by Chainlink to the standardized 10^{18} decimals (wei units), thus ensuring consistency.

Step 2: Collateral Price in USDC The current incorrect implementation in the contract is:

$$\text{CollateralPriceInUsdc}_{(\text{Incorrect})} = \frac{\text{CollateralPriceInUsd} \times \text{usdcPrice}}{10^{\text{usdcChainlinkDecimals}}}$$

Error Identification: This calculation multiplies two USD-denominated values, leading to a unit inconsistency:

$$\text{USD} \times \text{USD} \neq \text{USDC}$$

Corrected Calculation: To achieve correct unit alignment, the formula must divide the collateral USD price by the USDC/USD rate, resulting in the following:

$$\text{CollateralPriceInUsdc}_{[\text{Correct}]} = \frac{\text{CollateralPriceInUsd} \times 10^{\text{usdcChainlinkDecimals}}}{\text{usdcPrice}}$$

This accurately represents the conversion from collateral USD valuation into USDC.

3. Formal Proof of Correctness

Invariant Definition: Let the collateral token have a USD price $P_{collateral/USD}$, and let USDC have a USD price $P_{USDC/USD}$. The invariant required by the system is:

$$P_{collateral/USDC} = \frac{P_{collateral/USD}}{P_{USDC/USD}}$$

Correct Implementation Proof: Given:

$$P_{collateral/USD} = \frac{\text{collateralPrice} \times 10^{18}}{10^{\text{collateralChainlinkDecimals}}}$$

Then the correct implementation is:

$$P_{collateral/USDC} = \frac{P_{collateral/USD} \times 10^{\text{usdcChainlinkDecimals}}}{P_{USDC/USD}}$$

Expanding fully, we have:

$$P_{collateral/USDC} = \frac{\left(\frac{\text{collateralPrice} \times 10^{18}}{10^{\text{collateralChainlinkDecimals}}}\right) \times 10^{\text{usdcChainlinkDecimals}}}{P_{USDC/USD}}$$

Simplifying the expression shows consistent dimensional units, preserving the required invariant:

$$\frac{\text{Collateral (USD)}}{\text{USDC (USD)}} = \text{Collateral (USDC)}$$

Conclusion: Thus, the corrected implementation adheres to the economic and mathematical invariant required by the lending and borrowing logic of the contract, establishing formal correctness.

Impact:
As it has been demonstrated, the total amount of `collateralPriceInUsdc` is incorrectly calculated. This is affecting the `minCollateralTokenAmount` needed to call `borrow()` and `changeCollateralAmount`, possibly leading to a DOS or to the request of an amount of collateral higher than needed. It also affects `collateralTokenForLiquidator`, leading to the user receiving more/less collateral than the correct amount when a liquidation takes place.

Recommendation:
Correct the formula used for calculating `collateralPriceInUsdc`:

```
`uint256 collateralPriceInUsdc = (collateralPriceInUsd * (10 **
usdcChainlinkDecimals)) / uint256(usdcPrice);
```

Incorrect fee is taken from the users due to hardcoded token decimals, leading to loss of funds for the protocol or the users.

Description:
The `Pools.sol` smart contract implements a `_receiveUSDC` function which is used to charge fees from the users every time a new pool is created:

```
function _receiveUSDC(uint256 amount) private {
        USDC.transferFrom(msg.sender, address(this), amount);
    }
```

The amount of charged fees is hardcoded in the POOL_FEE variable:

```
uint256 constant POOL_FEE = 200_000000;
```

The protocol is going to be deployed to different blockchains: ETH, BASE, Polygon, BSC, Layer2 on OP stack (like OP). However, the USDC token decimals differs on each chain.

Impact:
The `POOL_FEE` variable is hardcoded to `200_000000`, which means that if USDC has 6 decimals (as in Ethereum) it will be 200 USDC tokens. However, USDC in BSC has 18 decimals which means that the protocol is receiving 0.0000000002 USC tokens as fee. The same scenario applies to the opposite case if the contract is deployed to a blockchain where USDC is less than 6 decimals, the protocol will receive more fee than expected.

Recommendation:
Set the token's decimals by constructor, or directly access them via interface and calculate teh total amount of fee depending on its decimals, instead of hardcoding the `POOL_FEE`.

Risk of Instant Liquidation and Front-Running After Protocol Resumption

Description:
The main protocol functions within ERC20CollateralPool.sol, including those used by users to repay their debt and avoid liquidations, can be paused. Additionally, liquidations themselves can also be paused.

Impact:
When liquidations are paused, users might believe their positions are safe. However, once the protocol resumes, users who were in a liquidable state prior to the pause can be immediately liquidated. Furthermore, since repayments are also paused, users may not have an opportunity to protect their positions. Upon resumption, they could be front-run by liquidators when trying to repay their debt, leading to unexpected losses.

Recommendation:
Implement a grace period after liquidations are resumed, allowing users time to repay their debt before liquidations become active again. This would help prevent instant liquidations and front-running scenarios.

Chainlink Data Staleness Threshold is Too Wide

Description:
The `_estimateERC20Value()` function within the `ERC20CollateralPool.sol` smart contract, which is used to return the collateral token value in USDC in WEI(1e18), currently allows Chainlink price data to be considered valid if it is up to **2 days old**:

```
(, int256 collateralPrice, , uint256 collateralUpdatedAt,) = pool
            .collateralDetails
            .collateralTokenChainlink
            .latestRoundData();

        (, int256 usdcPrice, , uint256 usdcUpdatedAt,) =
usdcChainlink.latestRoundData();

        if (collateralUpdatedAt < block.timestamp - 2 days) revert
ChainlinkDataTooOld();
        if (usdcUpdatedAt < block.timestamp - 2 days) revert
ChainlinkDataTooOld();
```

This is an excessively wide threshold, especially for volatile assets. Within 2 days, asset prices can change significantly, introducing severe pricing risks.

Impact:
Allowing stale prices up to 48 hours increases the risk of price manipulation, incorrect accounting of collateral, and loss of funds. If the price data is stale but still accepted by the system, users could exploit the outdated prices, resulting in inaccurate valuations of collateral and potentially leading to loss of funds in liquidations, deposits, or withdrawals. In rapidly moving markets, 2 days is an extremely long time for price data to be considered reliable.

Recommendation:
It is strongly recommended to reduce the allowable data age to, for example 1-2 hours, instead of 2 days. This ensures that only recent and reliable price data is used, significantly reducing the risk of stale price exploitation and improving the protocol's security against outdated price manipulation.

Attack can steal funds from the Staking contract when certain staking/reward tokens are used.

Description:
The `unstake()` function within the `Staking.sol` smart contract is used by the users to unstake their previously staked tokens and receive them back plus the rewards generated.

However, this function does not follow the CEI pattern and performs 2 external calls before updating the contract state:

```
plan.stakingToken.safeTransfer(msg.sender, userStake.amount);
        plan.rewardToken.safeTransfer(msg.sender, reward);

        userStake.claimed += reward;
        userStake.unstaked = true;
        plan.totalUnstaked += userStake.amount;
```

If any of the staking or reward tokens is a custom token that shifts the execution flow to the receiver address/contract, it would allow it to execute a reentrancy call before the contract state has been updated.

Impact:
If any of the staking or reward tokens is a custom token that shifts the execution flow to the receiver address/contract would allow it to recall the `claimRewards()` function, bypass the check and receive several rewards before `userStake.claimed` has been updated:

```
if (userStake.unstaked) revert StakeAlreadyUnstaked();
```

This issue has been addressed as High severity because the described scenario can only take place under the use of certain tokens, otherwise, if would have been set as Critical.

Recommendation:
Use `nonReentrant` modifier and update the `unstake()` function to follow the CEI pattern:

```
function unstake(uint256 _stakeIndex) external whenNotPaused nonReentrant
{
        Stake storage userStake = _getUserStake(msg.sender, _stakeIndex);

        if (userStake.unstaked) revert StakeAlreadyUnstaked();

        Plan storage plan = plans[userStake.planId];

        if (userStake.stakeTime + plan.lockDuration > block.timestamp)
            revert StakeIsLocked();

        uint256 reward = _calculateStakeReward(userStake,
uint48(block.timestamp));

     userStake.claimed += reward;
        userStake.unstaked = true;
        plan.totalUnstaked += userStake.amount;

        plan.stakingToken.safeTransfer(msg.sender, userStake.amount);
        plan.rewardToken.safeTransfer(msg.sender, reward);

        emit Unstaked(
            msg.sender,
            userStake.planId,
            userStake.amount,
            reward,
            _stakeIndex
        );
    }
```

Rewards Balance In The Staking Contract Is Not Considered Which May Result In Insolvency And Denial Of Withdrawals

Description:
The staking contract allows for users to deposit staking tokens against a staking plan to earn rewards over a period of time. Rewards for users are calculated based on the `amount * apy * timeWithRatio * raio / RATIO_DECIMALS_DIVIDER * PERCENTAGE_MULTIPLIER` over a period of 365 days (one year) however, the balance of rewards tokens in the contract is not factored into the calculations which could result in the contract going into an insolvent state.

Impact:
Should the staking contract go insolvent, the contract will not be able to payout rewards to users as there is a set amount of rewards prescribed to users without considering the reward token balance in the calculations for rewards. In addition to this, if the contract is insolvent, if users attempt to unstake the contract will revert resulting in stuck tokens as rewards tokens are also paid out when calling `unstake`. It is for this reason that this finding was rated a high in severity due to knock on effect.

Proof of Concept
The aforementioned scenario is demonstrated in the following foundry test:
https://gist.github.com/chris-zokyo/1bd0278b359d452d08ef6e65636805b4

Recommendation:
This can be resolved in one of the following two ways:

- Instead of paying out rewards to each user, pay out rewards to each staking token that is staked in the contract when calculating rewards so that the balance of rewards tokens will always be enough.
- Alternatively, if the protocol team is adamant on the current implementation, create a mapping which represents the “tokens owed” to users so that if the staking contract cannot distribute rewards tokens anymore, their owed balance is added to the mapping to be claimed at a later date when the contract is topped up by admins.

Finally, removing the rewards token transfer for both of the aforementioned points is also recommended to mitigate the risk of stuck tokens should the contract go insolvent.

Withdrawal Function In Staking Allows Default Admins To Rug The Contract

Description:
The staking contract allows for users to deposit their staking tokens into the contract and earn rewards over a period of time. There exists a `withdraw` function which allows a user with the default admin role to extract the balance of the contract into their address which lacks validation on the token being used.

Impact:
Should a member of the protocol team act maliciously or have their wallet compromised, through violation of access controls and segregation between owner powers and user funds, they would be free to withdraw the entire balance of the contract resulting in the rugging of user balances. This was rated a medium in severity because this relies on an edge case where a contract owner is compromised or behaves maliciously.

Recommendation:
It is recommended that there is sufficient validation through a require statement on the token parameter to ensure that it's not possible to withdraw staking tokens from the contract. This will create some additional access controls between user funds and the admin powers.

Hackers Note:

- This rug vector also exists in Pools.sol via the adminWithdrawAll function. It's recommended that there is some segregation between USDC and other collateral tokens in the contract and the powers of the contract owners - enough powers should be given in order to run day to day operations, but not be totally overpowered where they have sole control over user funds.

Collateral ERC20 Tokens With More Than 18 Decimals Are Incompatible In The ERC20CollateralPool

Description:
The `_calculateCollateralTokenAmount` function in the ERC20CollateralPool contract is responsible for calculating the collateral token amount from the USDC amount and LTV ratio. This equation extracts the decimals and subtracts 18 from the returned amount of decimal places; however, this does not consider the edge case of decimals being more than 18 places.

Impact:
Tokens with decimals higher than 18 will always revert the contract. This might include tokens such as NEAR token or YOCTO which are of 24 decimal places thus limiting the number of collateral tokens being compatible with the contracts current implementation.

Recommendation:
It's recommended that calculations for _calculateCollateralTokenAmount is updated to cater for tokens with decimals higher than 18

Non implemented checks can lead to unexpected behaviors.

Description:
The `addPlan()` function within the `Staking.sol` smart contract is used to create new plan for the staking. This function receives the `stakingToken´ and the `rewardToken`, however it is not checked if both tokens are different. The function also receives the `stakingEndTime` and `rewardEndTime`, however, it is neither checked if these times are higher than the current one:

```
function addPlan(
        IERC20Upgradeable stakingToken,
        IERC20Upgradeable rewardToken,
        uint256 maxStaked,
        uint256 minStakeAmount,
        uint256 initialRatio,
        uint48 stakingEndTime,
        uint48 rewardEndTime,
        uint48 lockDuration,
        uint16 apy,
        uint16 apyAfterUnlock
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (rewardEndTime < stakingEndTime + lockDuration) revert
RewardEndTimeTooLow();
        _checkIfPlanAlreadyExists(
            stakingToken,
            rewardToken,
            maxStaked,
            minStakeAmount,
            stakingEndTime,
            rewardEndTime,
            lockDuration,
            apy,
            apyAfterUnlock
        );
        plans.push(
            Plan(
                stakingToken,
```

```
                rewardToken,
                0,
                0,
                maxStaked,
                minStakeAmount,
                stakingEndTime,
                rewardEndTime,
                lockDuration,
                apy,
                apyAfterUnlock
            )
        );
        planTokenRatios[plans.length - 1].push(
            TokenRatio(
                uint48(block.timestamp),
                initialRatio
            )
        );

        emit PlanAdded(plans.length - 1, plans[plans.length - 1]);
    }
```

Impact:
If the parameters are wrongly set by error the function will not revert and will add a non desired plan to the staking contract.

Recommendation:
Implement the following checks:

```
require(stakingEndTime > block.timestamp, "Incorrect stakingEndTime");
require(rewardEndTime > block.timestamp, "Incorrect rewardEndTime");
```

CEI pattern not followed allowing reentrancy attacks under certain conditions

Description:
The `ERC20CollateralPool.sol` smart contract executes transfers to external addresses in several functions, either to transfer USDC to users or to handle collateral tokens. In some cases, these token transfers are performed before updating the full contract state (i.e., before calling `_updateRewards`).

Impact:
USDC is a well-known token; however, the protocol is designed to be deployed on multiple chains. Additionally, the collateral token used in the pools can be any token selected by the admin, including tokens that may trigger a fallback function upon transfer.
In such cases, the recipient of the transfer could gain control of the execution flow and potentially execute a reentrancy attack by re-entering the pool's contract before the full contract state has been updated.

Recommendation:
Review all functions that execute a transfer of USDC, and pay extra attention to those handling collateral token transfers. Ensure that the full contract state is updated before performing these transfers to prevent potential reentrancy attacks or incorrect accounting.

Sequencer Downtime Risk on L2 Chainlink Price Feeds

Description:
The `_estimateERC20Value()` function within the `ERC20CollateralPool.sol` smart contract, is used to return the collateral token value in USDC in WEI(1e18) by using Chainlink Price Feeds. As confirmed by the client, the protocol is going to be deployed to different blockchains, including L2 blockchains. For L2's, Chainlink Price Feeds rely on the L2 sequencer to relay price updates to the network. If the sequencer goes offline or becomes unavailable, price feeds will stop updating until the sequencer recovers. This creates a critical dependency between the oracle system and the sequencer's uptime.
Also consider that the call to `latestRoundData()` may revert. Consider adding a try-catch block if you want to handle the case where the revert takes place.

Impact:
If the sequencer is down, price feeds will freeze. The protocol will operate based on stale prices. This can lead to failed liquidations, incorrect margin calls, or the creation of bad debt. During periods of high volatility, this could result in cascading insolvencies across protocols on the affected L2.

Recommendation:
It should implement sequencer health checks to detect downtime and automatically revert or handle the case where the sequencer is down.

Reference:
Chainlink Documentation - L2 Sequencer Feeds
https://docs.chain.link/data-feeds/l2-sequencer-feeds

The transfer of certain token's transfer can silently fail.

Description:
The `withdraw()` function within the `ERC20LazyVesting.sol` smart contract is used by the admin to withdraw ERC20 funds from contract, if he announced the withdrawal.

```
function withdraw(IERC20[] calldata _tokens, address _recipient)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
    {
        require(_recipient != address(0), "Recipient is zero address");

        for (uint256 i = 0; i < _tokens.length; i++) {
            require(
                withdrawAnnouncmentTimestamps[_tokens[i]] != 0 &&
                withdrawAnnouncmentTimestamps[_tokens[i]] <
                (block.timestamp - 30 days),
                "Not enough time has pased"
            );

            uint256 balance = _tokens[i].balanceOf(address(this));

            emit TokensWithdrawn(_tokens[i], _recipient,
withdrawAnnouncmentTimestamps[_tokens[i]], balance);

            _tokens[i].transfer(_recipient, balance);
        }
    }
```

This function implements a `transfer()` of tokens which can silently fail without reverting, depending on the token transfer implementation function.

The same scenario is present within the `Pools.sol` smart contract and the transfer of tokens and also in `ERC20CollateralPool.sol'.

Impact:
If the transfer silently fails, it will not be noticed that the transfer has actually failed.

Recommendation:
Use the `safeTransfer()` function from the Openzeppelin's SafeERC20 library:

```
_tokens[i].safeTransfer(_recipient, balance);
```

A pool using a free-on-transfer token as collateral will result in incorrect accounting of collateral amounts.

Description:
The `ERC20CollateralPool.sol` smart contract allows users to borrow funds in exchange for a collateral amount. Each pool can be created with different assigned collateral tokens. When a user wants to borrow, they execute the `borrow()` function, specifying the `collateralTokenAmount` as a parameter. This parameter is used to transfer the specified amount of tokens to the contract and to account for the collateral deposited for the loan. However, if a transfer fee exists, the recorded collateral amount will be higher than the actual amount deposited, leading to inaccurate accounting.

```
pool.collateralDetails.collateralToken.safeTransferFrom(
            msg.sender,
            address(this),
            collateralTokenAmount
        );
```

The exact same scenario takes place within the `changeCollateralAmount()` function by indicating the `newCollateralTokenAmount` parameter.

Impact:
Consider the following scenario:

1. A token with a 10% fee is used.
2. The user indicates that he is using 100 tokens as collateral.
3. Only 90 tokens will be transferred to the contract as a result of the fee. However, the borrow object will account 100 tokens as collateral.

This issue makes liquidation more difficult, as users can provide a lower amount of funds as collateral than should be required. The severity of this issue has been classified as Medium because only the admin can create new pools and decide the collateral token used. Otherwise, it would have been a high-critical vulnerability.

Recommendation:
Track the balance before and after the transfer of the collateral amount and use the difference as the accounted amount for the borrowed object.

Developer comments:
Fee on transfer tokens will not be used.

The amount of staked funds is not decreased when a user unstakes, leading to an incorrect track of funds

Description:
The function `unstake()` within the `Staking.sol` smart contract is used by users to unstake their previously staked tokens on a certain plan and receive the rewards and the staked tokens back. This function increased the `totalUnstaked` variable for the plan but does not decreased the `totalStaked` one:

```
userStake.claimed += reward;
userStake.unstaked = true;
plan.totalUnstaked += userStake.amount;
```

The same scenario is present within the `restake()` function for the case of the `oldPlan.totalStaked`.

Impact:
If a user unstake his tokens, the `totalStaked` variable will reflect an incorrect amount of tokens.

Recommendation:
Add the following line to the `unstake` function:

```
plan.totalStaked -= userStake.amount;
```

Archiving A Pool In The Pools Contract Which Still Contains User Funds Will Result In Stuck Tokens

Description:
The Pools contract allows for users to create pools where other users can deposit into by committing into the pool in exchange for rewards in a lending type blueprint where collateral can be posted. The pool owner has the authority to create special transactions against these pools which includes closing or archiving a pool however, there is insufficient validation when archiving a pool which may result in stuck tokens.

Impact
Users who have committed to a pool and have forgotten to uncommit or have not uncommitted for another reason will lose their funds if the pool is archived, unable to withdraw for no valid reason.

Proof of Concept
The following proof of concept built in foundry outlines this scenario:

https://gist.github.com/chris-zokyo/c6032dd6477af22a2c78d502c05b20a1

Recommendation:
It's recommended that user funds are still made claimable by creating a functionality which accounts for how many tokens are owed to the user and allowing them to claim at a later date.

Collateral Token LTV Percentage Not Explicitly Validated Against Zero

Description:
The `collateralTokenLTVPercentage` parameter used in collateral calculations isn't explicitly validated against being set to zero. If mistakenly set to zero by the administrator, it could result in division-by-zero errors, causing unexpected contract failures or reverts during critical financial operations.

Impact:
Whilst this issue is low in severity, this still poses a security risk as human errors still occur in DeFI. Contract administrators who accidentally set a variable to a zero value will have an impact on the contract's day to day operations which is not limited to the creation of security issues.

Proof of Concept
An admin inadvertently announces and commits a pool edit with `collateralTokenLTVPercentage` set to `0`. Any subsequent calls to borrow or calculations relying on LTV percentage would revert due to division-by-zero, disrupting the pool's intended operation.

Recommendation:
Implement an explicit validation check to ensure `collateralTokenLTVPercentage` is strictly greater than zero during pool initialization

Cross-Site Scripting (XSS) via Token Symbol Injection

Description:
The `ERC20Factory` contract allows arbitrary strings for token names and symbols when deploying new ERC20 tokens. If a malicious user deploys a new token and inserts a JavaScript payload or malicious HTML into the token's symbol, it could trigger Cross-Site Scripting (XSS) vulnerabilities in frontend applications (such as block explorers, wallets, or protocol websites) that improperly render token metadata without sanitization or proper encoding.

Impact:
As anybody can create an ERC20 which may be displayed by the frontend, this can allow a malicious user to inject javascript code into the name and/or symbol elements of the token which can cause users accidentally to sign unexpected transactions.

Proof of Concept
A malicious attacker deploys a new ERC20 token through the `ERC20Factory` contract, setting the token symbol to a malicious payload, such as:
"><script>alert('XSS')</script>

When users or developers access a web-based interface or block explorer that fetches and directly renders token metadata (name, symbol) without proper encoding or validation, the malicious JavaScript executes in the victim's browser context, potentially stealing sensitive data or performing unauthorized actions.

Recommendation:

- Implement strict validation and sanitization of token metadata (name, symbol) before rendering it on any frontend or user interface. Always use proper encoding techniques to neutralize potential scripts.
- Consider enforcing character restrictions and length limitations on token metadata within the factory contract to reduce the risk of malicious payloads being included at the blockchain level.
- Ensure any frontend components fetching blockchain data sanitize and encode all token metadata before displaying to prevent browser-side attacks.

Unchecked Array Length Leading to Excessive Gas Costs

Description:
The `createPool` function allows unbounded arrays `(collateralTokens)`. A malicious or careless user could input an excessive number of collateral tokens, causing a significant increase in gas usage or even transaction failure due to exceeding block gas limits.

Impact:
Should the conditions of this case be met, this will result in excessive gas usage resulting in out of gas errors.

Proof of Concept

- A user attempts to create a pool with an excessively large collateral array.
- The transaction either fails due to exceeding gas limits or incurs unexpectedly high transaction fees.

Recommendation:

- Impose an explicit upper limit on the number of collateral tokens allowed per pool creation.
- Clearly document these constraints in the function specifications.

Developer comments:
We have no way of knowing what the maximum number of collateralTokens users will use. So would leave this as is

ERC20CollateralPoolStorage initialization will revert due to token decimal not matching

Description:
The `__ERC20CollateralPoolStorage_init()` function within the `ERC20CollateralPool.storage.sol` smart contract implements the following 'if statement':

```
if (_usdc.decimals() != 6) revert InvalidUSDCAddress();
```

The team has confirmed that the protocol is going to be deployed to different blockchains: 'ETH, BASE, Polygon, BSC, Layer2 on OP stack (like OP)'. The `usdc` token is not 6 decimals in every desired blockchain.

Impact:
The contract will not be able to get initialized due to `usdc` being not 6 decimals for certain blockchains.

Recommendation:
Remove the 'if statement' and be really cautious when initializing the smart contract for setting the correct `usdc` token.

Using the deprecated `_setupRole()` function may lead to unexpected behaviors.

Description:
The `constructor` of the `ERC20LazyVesting.sol` smart contract calls `setupRole()`. However, this function is deprecated as stated within the `AccessControl.sol` smart contract:

```
* NOTE: This function is deprecated in favor of {_grantRole}.
     */
    function _setupRole(bytes32 role, address account) internal virtual {
        _grantRole(role, account);
    }
```

The same scenario is present within the `constructor()` of the `Staking.sol`, `Pools.sol` smart contract.

Impact:
Using a deprecated function may lead to unexpected behaviors.

Recommendation:
Use `_grantRole()` instead of `_setupRole()`:

```
constructor(address _admin, address _operator) {
        _setRoleAdmin(OPERATOR_ROLE, DEFAULT_ADMIN_ROLE);
        _grantRole(DEFAULT_ADMIN_ROLE, _admin);
        _grantRole(OPERATOR_ROLE, _operator);
    }
```

Developer comments:
`setupRole()` works as `grantRole()`.

INFORMATIONAL-2 | ACKNOWLEDGED

Some actions are centralized

Description:
Some actions are centralized, depending on the execution of a single central authority:

- `ERC20.sol`: Tokens can only be transferred if they are not paused. The owner is the one responsible for pausing/unpausing tokens. Only the owner can also burn tokens.

- `Staking.sol`: Admin has the possibility to pause the contract and not allow unstake of tokens or claim rewards. Admin has the possibility of withdrawing any token from the contract. Admin is able to modify any plan at any time.
- `Pools.sol`: Deployer can pause contract and not allow pool owners and users execute certain actions. Admin can drain the whole contract by executing `adminWithdrawAll()`.

Developer comments:
This is by design.

INFORMATIONAL-3 | RESOLVED

Variable with incorrect spelling

Description:
There is a `withdrawAnnouncmentTimestamps` variable within the `ERC20LazyVesting.sol` smart contract which is wrongly written.

Recommendation:
Fix the spelling: `withdrawAnnouncementTimestamps`.

It is an engineering best practice using Openzeppelin libraries.

Description:
The `ERC20LazyVesting.sol` smart contract implements several functions related to merkle trees and merkle proofs. However, it is recommended to better use the already implemented functions by Openzeppelin.

Recommendation:
Consider using `Merkle.sol` from Openzeppelin: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/MerkleProof.sol

Developer comments:
We will use our implementation

uint256 can never be lower than 0

Description:
The `_requireAmountBiggerThanZero()` function within `ERC20CollateralPool.sol` smart contract checks if `uint256 amount` is lower of equal than 0.

```
function _requireAmountBiggerThanZero(uint256 amount) private pure {
        if (amount <= 0) revert AmountCannotBeZero();
}
```

An `uint256` variable can never be lower than 0.

Impact:
There is no security impact here, that's why this issue has been set as informational severity, but it is recommended to follow best engineering practices.

Recommendation:
Update the function:

```
function _requireAmountBiggerThanZero(uint256 amount) private pure {
        if (amount == 0) revert AmountCannotBeZero();
}
```

INFORMATIONAL-6 (ENGINEERING PRACTICES) | ACKNOWLEDGED

Hashes Can Be Used To More Efficiently Check If A Staking Plan Exists In The Staking Contract

Description:
The staking contract uses `_checkIfPlanAlreadyExists` in order to validate if a plan has already been added to the contract. This uses a for loop which validates each attribute of every plan which is done in O(n) time.

Recommendation:
It's recommended that this can be done more efficiently by hashing the entire plan and inserting it into a mapping which points to a boolean value. A plan can be checked by rehashing the passed plan parameters and asserting whether the value points to true which is done in O(1) constant time.

Developer comments:
Acknowledged.

INFORMATIONAL-7 (ENGINEERING PRACTICES/CODE PRESENTATION) | RESOLVED

Unused Code In Staking.storage.sol

Description:
The `__StakingStorage_init` function in StakingStorage contains unused code which is commented out.

Recommendation:
It's recommended that these comments are removed from the contract to better increase code presentation.

	./erc-20-mixer/ERC20Mixer.interface.sol **./erc-20-mixer/ERC20MixerFactory.sol** **./erc-20-mixer/ERC20Mixer.sol** **./data-storage/DataStorage.sol** **./data-storage/IDataStorage.sol** **./pools/Pools.storage.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)	Fail

Key Variables Are Not Returned In Certain Functions

Description:
The `createPool` in the Pools function is a void function which does not return anything which makes it more difficult for users who integrate with the contracts to obtain the pool id.

In addition to this, the ERC20Factory shares a similar issue in the `createERC20Token` function where the address is not returned for the deployed token.

Recommendation:
It is recommended that the address or the integer that is created is returned on function completion to more easily obtain the pool identifier.

	./pools/Pools.interface.sol **./pools/Pools.sol** **./erc-20/ERC20Factory.sol** **./erc-20/ERC20.sol** **./vesting/ERC20LazyVesting.interface.sol** **./vesting/ERC20LazyVesting.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)	Fail

	./staking/Staking.interface.sol **./staking/Staking.storage.sol** **./staking/Staking.sol** **./erc-20-collateral-pool/ERC20CollateralPool.storage.sol** **./erc-20-collateral-pool/AggregatorV3Interface.sol** **./erc-20-collateral-pool/ERC20CollateralPool.interface.sol** **./erc-20-collateral-pool/ERC20CollateralPool.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)	Fail

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

zokyo