Cyberscope

Audit Report

ElectraProject

December 2023

Repository https://github.com/dashewski/ElectraProject

Commit ff27fef894ac907ba6ea3d22b300b6917ebbcb87

Audited by © cyberscope

Table of Contents

Review

Repository	https://github.com/dashewski/ElectraProject
Commit	ff27fef894ac907ba6ea3d22b300b6917ebbcb87

Audit Updates

Initial Audit	12 Dec 2023

Source Files

Filename	SHA256
Treasury.sol	6f8002899da0dc3ddfec2faf5ecd82a757af28519df037edfc72032fb48bdf0f
Item.sol	41f58b9e16150a2056536d0949c77131fddf9ee976c446dc49f20b147e90697e
FlexStakingStrategy.sol	6516b08f78c73f1bdcf8859bca7e42fc5eef5066a690c72d22bf44af782a4a25
FixStakingStrategy.sol	29a80dde401e0b0eda7fb0577a1c5369ea6283c7c8c1e935012ef39cdea4db59
AddessBook.sol	aca346c9da718aea0a984e0e4615f8a67e9c2d1451941ab06c82aa55d90c8b29
utils/imports.sol	4d0efb996c7047b78adde86b580ffa8e1a706bb069a0a1fbfb393f9de838af21
utils/Pricer.sol	82064f550fd8bcdfa408abd9c47012574a5f4388354ba9135d6df7a11e4899e6
utils/DateTimeLib.sol	e8477f7e09290d5d5f7d6ade77e8c0971692957872061a12c0f2b618f5ccd0af

interfaces/ITreasury.sol	0bb6e7d746fa95f6747153841ffdba6e04c38a8555a16713ae8a656772b0156e
interfaces/IStakingStrategy.sol	55bf8c8d1ccd2f8a3b43b49afef2f0bd58349d7a2a5d0c935d85ad0138f1f368
interfaces/IPricer.sol	1e873e015dcb42eb261bf27b41c67d52c7f4f5c50933ac61de9e2b529b337b4b
interfaces/IItem.sol	1d50fbca0972d7252a131c92988ad4012b1b57e98a6d6aa58b77066fa8756992
interfaces/IFlexStakingStrategy.sol	23b7fec56d8e651428592ab61016b937a285d314c67cb36aaea8d7c355d2b522
interfaces/IFixStakingStrategy.sol	ae10a233fe01aea428a27d3c28e497beaa3d43a9397a53ec1627a1b7020083c9
interfaces/IAddressBook.sol	94f3dca12093c7ab8ee2b1acd15d56408e12313c374e4b3704660a315588854e
core/Treasury.sol	4bc7466bde1856bd15571e4416892c3c06d428360b97d5b3d32cf54090901d85
core/Item.sol	2417e5fe6de0cd6bcc1d1d0fb1aea43b8c6e6a8677a334042b6054d2f251d2c1
core/AddessBook.sol	362531bff3483a2c8c96dbd6b8f43f53fe76c6ab99a620783b7692102831ebec

Overview

AddressBook Contract

The contract handles the management of a set of items and staking strategies, with a focus on administrative control and upgradeability. The contract is structured with a designated 'product owner' who holds the exclusive authority to modify key aspects of the contract, including the ownership itself, the addition or removal of items, and the management of staking strategies. This contract is designed for dynamic management and control of a specific set of resources or activities within the decentralized application, with a strong emphasis on the role of the product owner in governing these activities.

Treasury Contract

The contract is designed to manage financial aspects, primarily focusing on token management and pricing. It includes mechanisms for adding, updating, and deleting token pricers, each associated with a specific token. The contract also features a withdrawal function, which is gated by ownership and governance checks, indicating a robust control mechanism over financial operations. The contract emits events for withdrawals, enhancing its transparency. Additionally, it includes a function to convert USD amounts to token amounts based on the latest price data, showcasing its capability in handling real-time financial data. Overall, the Treasury contract seems to be a critical component in a decentralized finance (DeFi) ecosystem, handling token pricing and treasury management with an emphasis on governance and upgradeability.

Item Contract

The Item contract is designed for creating and managing a unique collection of NFTs (Non-Fungible Tokens), with an integrated financial and staking system. At its core, the contract allows users to mint NFTs through its `mint` function, where users pay with a specific token. The payment is converted to USD equivalent and sent to a treasury. Upon payment, an NFT is minted and assigned a unique ID, and simultaneously, it is staked according to a predefined staking strategy. This integration of minting and staking within the same transaction streamlines the user experience, making the contract a multifaceted tool for NFT creation, financial transactions, and asset management in a decentralized environment.

FixStakingStrategy Contract

Stake Functionality

The 'FixStakingStrategy' contract introduces a unique staking functionality, primarily initiated during the NFT minting process in the associated `Item` contract. When an NFT is minted, it is automatically staked through the stake function. This function records the staking status of each NFT, along with its initial and final timestamp, based on a predetermined rewards period and lock years. It also captures the price of the item at the time of staking, ensuring accurate financial tracking. The staking action is concluded by emitting a `Stake` event, detailing key information like the item address, token ID, owner, price, and staking timeframes. This process not only secures the NFT in a staking mechanism but also sets the stage for future rewards and interactions within the contract's ecosystem.

Claim Functionality

The `claim` function in the contract offers users the ability to claim rewards for their staked NFTs. This function ensures that only the NFT owner can initiate the claim. It calculates the rewards based on the staked period and the price of the NFT, leveraging the `estimateRewards` function. The estimated rewards are then converted into a withdrawable token amount, which is withdrawn from the treasury and transferred to the NFT owner. The function meticulously tracks the number of claimed periods and the total amount of rewards withdrawn. This mechanism not only incentivizes users to stake their NFTs but also ensures a fair and transparent process for reward distribution.

Sell Functionality

The `sell` function in the contract allows users to sell their staked NFTs, subject to certain conditions. It verifies that the NFT owner is the one initiating the sale and that the NFT is eligible for sale based on the contract's `canSell` criteria. This eligibility is determined by checking if the current timestamp is beyond the final staking timestamp and if all possible reward periods have been claimed. The `estimateSell` function calculates the sell amount by considering the NFT's initial price and applying a depreciation rate based on the lock years. The amount receivable by the user is then converted into a withdrawable token amount, and the NFT is burned, concluding its lifecycle. This sell functionality adds a layer of flexibility and exit strategy for users, aligning with the dynamic nature of NFT markets.

FlexStakingStrategy Contract

Stake Functionality

The 'FlexStakingStrategy' contract introduces an advanced staking mechanism for NFTs. Its `stake` function is integral to this process, setting the stage for flexible staking durations and dynamic rewards. When an NFT is staked, the function records its staking status, initial, start, and final timestamps, and calculates the price of the item. The function also handles the division of staking periods into initial months and remaining months, accommodating different rewards rates and depreciation calculations. This flexibility allows for a tailored staking experience, adapting to various user preferences and market conditions. The contract emits a `Stake` event, encapsulating all relevant details, enhancing transparency and traceability in the staking process.

Claim Functionality

The `claim` function in the 'FlexStakingStrategy' contract enables users to claim rewards for their staked NFTs. This process involves a complex calculation of rewards based on the duration of the stake, the price of the NFT, and the earnings set for each period. The rewards are then converted into a withdrawable token amount and transferred from the treasury to the NFT owner. The contract meticulously records the number of claimed periods and the total rewards withdrawn, ensuring accuracy and fairness in the distribution of rewards. This dynamic rewards system incentivizes users for longer staking periods, aligning with the flexible nature of the contract.

Sell Functionality

The `sell` function allows NFT owners to sell their staked assets, provided certain conditions are met. This functionality is designed to account for the flexible staking periods, adjusting the sellable amount based on the total time staked and depreciation rates. The contract calculates the sell price, considering the time elapsed and the original price of the NFT, and then facilitates the transfer of the equivalent token amount from the treasury to the owner. The contract ensures that the NFT is burnt after the sale, marking the end of its lifecycle within the staking strategy.

Product Owner Functionalities

The 'FlexStakingStrategy' contract also includes specific functionalities for the contract owner, emphasizing governance and adaptability. These include the ability to set earnings for different periods, reflecting the flexibility in reward distribution, and the ability to update deposits, which is critical for maintaining accurate financial records within the staking strategy. The owner can adjust parameters like lock years, rewards rates, and depreciation rates, allowing for strategic management of the staking ecosystem. These functionalities underscore the owner's role in overseeing the contract's operation, ensuring it remains responsive to changing market dynamics and stakeholder needs.

Staking Strategies Differences

The 'FixStakingStrategy' and 'FlexStakingStrategy' represent two distinct approaches to staking within the NFT ecosystem. The 'FixStakingStrategy' is characterized by its straightforward and predictable staking process, where NFTs are staked for a fixed period with predetermined rewards and a clear end date. This strategy offers stability and simplicity, appealing to users who prefer a set-and-forget type of staking. In contrast, the 'FlexStakingStrategy' introduces a more dynamic and adaptable staking model. It allows for variable staking durations and offers a more complex rewards system that can change based on different time periods and market conditions. This strategy caters to users seeking flexibility and the potential for optimized returns, as it adjusts to various user preferences and market dynamics.

Roles

Owner

The product owner which is set by the owner through the `setProductOwner` can interact with the following functions:

- function addItem(address _item)
- function deleteItem(address _item)
- function addStakingStrategy(address _stakingStrategy)
- function deleteStakingStrategy(address _stakingStrategy)
- function setTreasury(address _treasury)
- function addToken(address _token, address _pricer)
- function updateTokenPricer(address _token, address _pricer)
- function deleteToken(address _token)
- function withdraw(address _token, uint256 _amount, address _recipient)
- function stopSell()
- function setNewMaxSupply(uint256 _maxSupply)
- function setBaseUri(string calldata _uri)
- function setEarnings(uint256 _year, uint256 _month, uint256 _formatedEarning)
- function updateDeposits()

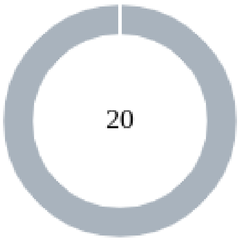
Users

The users can interact with the following functions:

- function mint(address _stakingStrategy,address _payToken,bytes memory _payload)
- function _baseURI()
- function claim(address _itemAddress,uint256 _itemId,address _withdrawToken)
- function sell(address _itemAddress,uint256 _itemId, address _withdrawToken)
- function stakingType()
- function canSell(address _itemAddress, uint256 _itemId)
- function estimateSell(address _itemAddress, uint256 _itemId)
- function claimTimestamp(address _itemAddress,uint256 _itemId,uint256 _monthsCount)
- function currentPeriod()
- function getAllExpiredMonths(address _itemAddress,uint256 _itemId)

- function estimateRewards(address _itemAddress,uint256 _itemId)

Findings Breakdown



- Critical 0
- Medium 0
- Minor / Informative 20

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	20	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Minor / Informative	MC	Missing Check	Unresolved
Minor / Informative	EDV	Excessive Depreciation Value	Unresolved
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor / Informative	MEM	Misleading Error Messages	Unresolved
Minor / Informative	RVL	Redundant Validation Logic	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	IEC	Inaccurate Earnings Calculation	Unresolved
Minor / Informative	IDH	Inconsistent Decimals Handling	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	MSC	Missing Sanity Check	Unresolved
Minor / Informative	IMU	Inconsistent Mapping Usage	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	RSW	Redundant Storage Writes	Unresolved

●	MEE	Missing Events Emission	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved

MC - Missing Check

Criticality	Minor / Informative
Location	FixStakingStrategy.sol#L228 FlexStakingStrategy.sol#L102,361
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically the `_monthsCount` variable should be greater than zero and the `initialMonths` value should be less than the `minMonthsCount` value.

```
    function initialize(
        address _addressBook,
        uint256 _minLockYears,
        uint256 _maxLockYears,
        uint256 _initialMonths,
        uint256 _initialRewardsRate,
        uint256 _yearDeprecationRate
    ) public initializer {
        addressBook = _addressBook;
        minLockYears = _minLockYears;
        maxLockYears = _maxLockYears;
        initialMonths = _initialMonths;
       ...
       minMonthsCount = 12 * _minLockYears + 1;
    }

    function claimTimestamp(
        address _itemAddress,
        uint256 _itemId,
        uint256 _monthsCount
    ) external view returns (uint256) {
        uint256 _finalTimestamp = finalTimestamp[_itemAddress][_itemId];
        uint256 _initialTimestamp = initialTimestamp[_itemAddress][_itemId];
        uint256 _nextClaimTimestamp = _initialTimestamp + _monthsCount *
REWARDS_PERIOD;
        if (_nextClaimTimestamp > _finalTimestamp) _nextClaimTimestamp =
_finalTimestamp;
        return _nextClaimTimestamp;
    }


    function claimTimestamp(
        address _itemAddress,
        uint256 _itemId,
        uint256 _monthsCount
    ) external view returns (uint256) {
        ...
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

EDV - Excessive Depreciation Value

Criticality	Minor / Informative
Location	FixStakingStrategy.sol#L221 FlexStakingStrategy.sol#L445
Status	Unresolved

Description

The contract is currently utilizing a `yearDeprecationRate` to calculate the depreciation of NFTs over time, which is deducted from the original `_itemsPrice`. This mechanism is implemented in the `estimateSell` function, where the deprecation is calculated based on the product of `_itemsPrice`, time elapsed, and `yearDeprecationRate`, divided by `10,000`. However, a significant issue arises if the contract owner sets an excessively high value for the `yearDeprecationRate`. In such scenarios, the calculated deprecation can exceed the original `_itemsPrice`, resulting in the function returning a zero value. This effectively prevents users from being able to sell their NFTs, as the estimated sell price becomes zero, thereby impacting the usability and fairness of the contract.

```
    function estimateSell(address _itemAddress, uint256 _itemId) public
view returns (uint256) {
        uint256 _itemsPrice = itemsPrice[_itemAddress][_itemId];
        uint256 deprecation = (_itemsPrice * lockYears *
yearDeprecationRate) / 10000;
        if (deprecation > _itemsPrice) return 0;
        return _itemsPrice - deprecation;
    }
```

```
    function estimateSell(address _itemAddress, uint256 _itemId) public
view returns (uint256) {
        ...
        uint256 _itemsPrice = itemsPrice[_itemAddress][_itemId];
        uint256 deprecation = (_itemsPrice * allExpiredMonths *
yearDeprecationRate) / 12 / 10000;

        if (deprecation > _itemsPrice) return 0;
        return _itemsPrice - deprecation;
    }
```

`

Recommendation

It is recommended to implement a safeguard mechanism within the contract to prevent setting the `yearDeprecationRate` to a level that would render NFTs unsellable. This could involve setting a maximum limit for the `yearDeprecationRate` or introducing a validation check within the `estimateSell` function to ensure that the deprecation does not exceed a certain percentage of the `_itemsPrice`. These measures will ensure that the depreciation mechanism functions as intended without unfairly penalizing users, thereby maintaining the integrity and attractiveness of the NFT marketplace.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	FixStakingStrategy.sol#L144,146 FlexStakingStrategy.sol#L269,278
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

Specifically the actual `withdrawTokenAmount` that is processed by the `_treasury` contract could differ from the `withdrawTokenAmount` that is emitted by the `Claim` event.

```
        ITreasury(_treasury).withdraw(_withdrawToken,
withdrawTokenAmount, msg.sender);

        emit Claim(
            _itemAddress,
            _itemId,
            _itemOwner,
            rewards,
            claimedPeriods,
            _withdrawToken,
            withdrawTokenAmount
        );
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

`Actual Transferred Amount = Balance After Transfer - Balance Before Transfer`

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	FixStakingStrategy.sol#L134 FlexStakingStrategy.sol#L258
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(rewards > 0, "rewards!");
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

RVL - Redundant Validation Logic

Criticality	Minor / Informative
Location	Treasury.sol#L72,81,120
Status	Unresolved

Description

The contract uses a `require` statement to check if the price of the token address is equal to the zero address. This check is performed by the line `require(pricers[_token] != address(0), "Treasury: not exists!");`. However, this validation is already encapsulated within the `enforceIsSupportedToken` function, which also includes the same require statement. This results in an unnecessary duplication of the same logic, leading to inefficiencies in the contract's execution.

```
   require(pricers[_token] != address(0), "Treasury: not exists!");

    function enforceIsSupportedToken(address _token) external view {
        require(pricers[_token] != address(0), "Treasury: unknown
token!");
    }
```

Recommendation

It is recommended to leverage the existing `enforceIsSupportedToken` function for validating the token address instead of repeatedly using separate `require` statements. This approach will not only streamline the contract by removing redundant code but also enhance the maintainability and clarity of the contract. Consolidating the validation logic into a single function ensures that any future updates or modifications to the validation criteria are centralized, reducing the risk of inconsistencies and potential errors in the contract's behavior.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	AddessBook.sol#L30 Item.sol#L76 FixStakingStrategy.sol#L131,164
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
enforceIsProductOwner(msg.sender);

IAddressBook(addressBook).enforceIsProductOwner(msg.sender);

_enforceIsStakedToken(_itemAddress, _itemId);
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

IEC - Inaccurate Earnings Calculation

Criticality	Minor / Informative
Location	FlexStakingStrategy.sol#L442
Status	Unresolved

Description

The contract is currently designed to iterate through a set of periods (months) to calculate rewards based on `earnings` mapping. However, there is a critical implementation detail where the `break` keyword is used within the loop. This keyword is triggered when the earnings for the current year and month mapping are zero. The use of `break` in this context prematurely terminates the loop, which can lead to an oversight in calculating rewards for subsequent periods. Specifically, even if the earnings for the current month are zero, it does not necessarily imply that the earnings for the following months will also be zero, since the product owner has the authority to set the earnings through the `setEarnings` function. Consequently, by breaking the loop, the contract neglects to consider the earnings set for the next `_claimedPeriodsCount` months, potentially leading to an inaccurate calculation of rewards.

```
        for (uint256 i = _claimedPeriodsCount; i < allExpiredMonths; ++i)
{
            ...

            (uint256 year, uint256 month, ) =
DateTimeLib.timestampToDate(
                DateTimeLib.addMonths(_startStakingTimestamp, i)
            );

            uint256 _earnings = earnings[year][month];
            if (_earnings == 0) {
                if (claimedPeriods_ == 0) rewards_ = 0;
                Break;
    }
```

Recommendation

It is recommended to replace the `break` statement with a `continue` statement in the loop. By doing so, the loop will skip the current iteration when it encounters a month with zero earnings but will continue to evaluate the earnings for subsequent months. This change ensures that all months within the `_claimedPeriodsCount` are taken into consideration for reward calculations, even if some months within the range have zero earnings. This adjustment will provide a more accurate and fair calculation of rewards, reflecting the actual earnings over the entire period considered.

IDH - Inconsistent Decimals Handling

Criticality	Minor / Informative
Location	FlexStakingStrategy.sol#L145 Treasury.sol#L17
Status	Unresolved

Description

The contract is currently implementing the `setEarnings` function to set earnings by multiplying the `_formatedEarning` value with `1e18`. This multiplication is intended to align the earnings with the 18-decimal format used by the USD token in the withdrawal process. However, the approach hardcodes the decimal value in the contract, rather than dynamically retrieving it from the treasury contract, where `USD_DECIMALS` is already defined as `18`. This hardcoded approach limits the contract's flexibility and adaptability to changes in the decimal structure of the associated USD token.

```
    function setEarnings(uint256 _year, uint256 _month, uint256
_formatedEarning) external {
        IAddressBook(addressBook).enforceIsProductOwner(msg.sender);
        ...
        uint256 _earnings = _formatedEarning * 1e18;
        earnings[_year][_month] = _earnings;

        ...
    }
    ...
    uint256 public constant PRICERS_DECIMALS = 8;
```

Recommendation

It is recommended to modify the `setEarnings` function to dynamically fetch the `USD_DECIMALS` value from the treasury contract instead of using the hardcoded `1e18` multiplier. This change will make the contract more adaptable to potential changes in the USD token's decimal structure and ensure consistency across the ecosystem. By retrieving the decimals directly from the source, the contract will enhance its resilience to updates or modifications in token specifications, thereby future-proofing its functionality and maintaining alignment with the broader system architecture.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	AddessBook.sol#L42 Treasury.sol#L59,85 Item.sol#L124FixStakingStrategy.sol#L76 FlexStakingStrategy.sol#L135
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

The contract is designed with a high degree of centralization, granting the contract owner extensive control over critical parameters. This centralized control poses significant risks to the contract's integrity and trustworthiness. Specifically, the contract owner has the authority to manipulate key aspects of the NFT staking ecosystem. These include the ability to set the NFT collection, add, delete, or withdraw tokens used for payments to users during `claim` or `sell` functions. Additionally, the owner can determine the maximum supply of NFTs that can be minted, halt the minting process at will, set reward rates, adjust the depreciation of the price, and define the duration of staking periods. Such centralized control can lead to potential misuse or arbitrary changes that may not align with the interests of the users or the community, thereby undermining the decentralized nature of the blockchain.

```
 function addItem(address _item) external {
        enforceIsProductOwner(msg.sender);
        items[_item] = true;
    }

    function deleteItem(address _item) external {
        enforceIsProductOwner(msg.sender);
        delete items[_item];
    }

    function addToken(address _token, address _pricer) external {
        IAddressBook(addressBook).enforceIsProductOwner(msg.sender);
        require(pricers[_token] == address(0), "Treasury: already
exists!");
        require(_pricer != address(0), "Treasury: pricer == 0");
        require(IPricer(_pricer).decimals() == PRICERS_DECIMALS,
"Treasury: pricer decimals != 8");
        pricers[_token] = _pricer;
    }

    function updateTokenPricer(address _token, address _pricer) external
{
        IAddressBook(addressBook).enforceIsProductOwner(msg.sender);
        require(pricers[_token] != address(0), "Treasury: not exists!");
        require(_pricer == address(0), "Treasury: pricer == 0");
        require(IPricer(_pricer).decimals() == PRICERS_DECIMALS,
"Treasury: pricer decimals != 8");
        pricers[_token] = _pricer;
    }

    function withdraw(address _token, uint256 _amount, address
_recipient) external {
    ...
    }
      function stopSell() external {
        IAddressBook(addressBook).enforceIsProductOwner(msg.sender);
        maxSupply = totalMintedAmount;
    }

    function setNewMaxSupply(uint256 _maxSupply) external {
        IAddressBook(addressBook).enforceIsProductOwner(msg.sender);
        require(_maxSupply > totalMintedAmount, "max supply less!");
        maxSupply = _maxSupply;
    }
 function initialize(
        address _addressBook,
        uint256 _rewardsRate,
        uint256 _lockYears,
        uint256 _yearDeprecationRate
    ) public initializer {
```

```
            ...
    }
        function setEarnings(uint256 _year, uint256 _month, uint256
_formatedEarning) external {
            ...
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

MSC - Missing Sanity Check

Criticality	Minor / Informative
Location	FlexStakingStrategy.sol#L102
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

The `minLockYears` variable should be less than the `maxLockYears` variable.

```
   function initialize(
        address _addressBook,
        uint256 _minLockYears,
        uint256 _maxLockYears,
        uint256 _initialMonths,
        uint256 _initialRewardsRate,
        uint256 _yearDeprecationRate
    ) public initializer {
        addressBook = _addressBook;
        minLockYears = _minLockYears;
        maxLockYears = _maxLockYears;
        ...

        ...

        maxMonthsCount = 12 * _maxLockYears + 1;
        minMonthsCount = 12 * _minLockYears + 1;
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. Specifically, the `_minLockYears` value should be checked to be less than `_maxLockYears` value.

IMU - Inconsistent Mapping Usage

Criticality	Minor / Informative
Location	FixStakingStrategy.sol#L137,176 FlexStakingStrategy.sol#L260,321
Status	Unresolved

Description

The contract is designed to handle the sale of items by calculating the `sellAmount` using the `estimateSell` function. This function determines the value by considering the original price paid for the item, reduced by its depreciation. However, a critical observation in the contract's logic is the use of the `withdrawnRewards` mapping. This mapping is employed to store reward amounts as calculated in the `claim` function. The issue arises because the `withdrawnRewards` mapping is also used to store the `sellAmount` within the `sell` function, which represents the price returned to the user after selling the item. Consequently, this mapping does not exclusively represent rewards as its name suggests, but also includes the value returned to users from sales. This dual usage of the `withdrawnRewards` mapping for distinct purposes, storing both rewards and return values from sales, can lead to confusion and misinterpretation of the data it holds.

```
    function claim(
        address _itemAddress,
        uint256 _itemId,
        address _withdrawToken
    ) external nonReentrant {
        ...

        (uint256 rewards, uint256 claimedPeriods) =
estimateRewards(_itemAddress, _itemId);
        require(rewards > 0, "rewards!");
        ...
        withdrawnRewards[_itemAddress][_itemId] += rewards;
        ...
    }

    function sell(
        address _itemAddress,
        uint256 _itemId,
        address _withdrawToken
    ) external nonReentrant {
        ...
        uint256 sellAmount = estimateSell(_itemAddress, _itemId);
        ...
        withdrawnRewards[_itemAddress][_itemId] += sellAmount;
```

Recommendation

It is recommended to reevaluate and potentially refactor the logic associated with the `withdrawnRewards` mapping. If the primary intent is to track only the rewards, then the contract should implement a separate mapping dedicated to storing the value returned to users during the sell function. This separation will enhance clarity and ensure that each mapping distinctly represents its intended data, one for rewards and another for the return value from sales. This approach will not only improve the readability and maintainability of the contract but also reduce the likelihood of errors or misunderstandings arising from the overlapping use of the `withdrawnRewards` mapping for two different financial concepts.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	FixStakingStrategy.sol#L133,167
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(msg.sender == _itemOwner, "only item owner!");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

CR - Code Repetition

Criticality	Minor / Informative
Location	Treasury.sol#L59
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically the `addToken` and `updateTokenPricer` share similar code segments.

```
    function addToken(address _token, address _pricer) external {
        IAddressBook(addressBook).enforceIsProductOwner(msg.sender);
        require(pricers[_token] == address(0), "Treasury: already
exists!");
        require(_pricer != address(0), "Treasury: pricer == 0");
        require(IPricer(_pricer).decimals() == PRICERS_DECIMALS,
"Treasury: pricer decimals != 8");
        pricers[_token] = _pricer;
    }

    function updateTokenPricer(address _token, address _pricer)
external {
        IAddressBook(addressBook).enforceIsProductOwner(msg.sender);
        require(pricers[_token] != address(0), "Treasury: not
exists!");
        require(_pricer == address(0), "Treasury: pricer == 0");
        require(IPricer(_pricer).decimals() == PRICERS_DECIMALS,
"Treasury: pricer decimals != 8");
        pricers[_token] = _pricer;
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	AddessBook.sol#L37
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
   function setProductOwner(address _newProductOwner) external {
        enforceIsProductOwner(msg.sender);
        productOwner = _newProductOwner;
    }

    function addItem(address _item) external {
        enforceIsProductOwner(msg.sender);
        items[_item] = true;
    }

    function deleteItem(address _item) external {
        enforceIsProductOwner(msg.sender);
        delete items[_item];
    }

    function addStakingStrategy(address _stakingStrategy) external {
        enforceIsProductOwner(msg.sender);
        stakingStrategies[_stakingStrategy] = true;
    }

    function deleteStakingStrategy(address _stakingStrategy)
external {
        enforceIsProductOwner(msg.sender);
        delete stakingStrategies[_stakingStrategy];
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	AddessBook.sol#L37
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
  function setProductOwner(address _newProductOwner) external {
        enforceIsProductOwner(msg.sender);
        productOwner = _newProductOwner;
    }

    function addItem(address _item) external {
        enforceIsProductOwner(msg.sender);
        items[_item] = true;
    }

    function deleteItem(address _item) external {
        enforceIsProductOwner(msg.sender);
        delete items[_item];
    }

    function addStakingStrategy(address _stakingStrategy) external {
        enforceIsProductOwner(msg.sender);
        stakingStrategies[_stakingStrategy] = true;
    }

    function deleteStakingStrategy(address _stakingStrategy) external {
        enforceIsProductOwner(msg.sender);
        delete stakingStrategies[_stakingStrategy];
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Treasury.sol#L42,54,59,69,79,89,108,119 Item.sol#L61,62,63,64,65,66,84,85,86,113,127,133 FlexStakingStrategy.sol#L103,104,105,106,107,108,134,180,252,282,356,362,363,364,374,375,388,389,438,445 FixStakingStrategy.sol#L77,78,79,80,97,125,126,127,158,159,160,199,200,215,221,229,230,231 AddessBook.sol#L25,37,42,47,52,57,62,73,77,81
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _addressBook
bool _value
address _token
address _pricer
address _recipient
uint256 _amount
uint256 _usdAmount
string calldata _name
string calldata _symbol
uint256 _price
uint256 _maxSupply
string calldata _uri
address _stakingStrategy
address _payToken

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	Treasury.sol#L93
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(
            (_addressBook.stakingStrategies(msg.sender) &&
onlyGovernanceWithdrawn == false) ||
                _addressBook.productOwner() == msg.sender,
            "Treasury: withdraw not authorized!"
        )
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	FlexStakingStrategy.sol#L199,200
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 ratio = (1e18 * initialDay) / (daysInStartMonth + 1)
uint256 _remainder = (_itemsPrice * ratio) / 1e18
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	FlexStakingStrategy.sol#L160
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 i
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Treasury.sol#L43Item.sol#L69 FlexStakingStrategy.sol#L110 FixStakingStrategy.sol#L82 AddessBook.sol#L26,39,66
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
addressBook = _addressBook
productOwner = _prodcutOwner
productOwner = _newProductOwner
treasury = _treasury
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
Treasury	Implementation	ITreasury, UUPSUpgradeable, MulticallUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	setOnlyProductOwnerWithdrawn	External	✓	-
	addToken	External	✓	-
	updateTokenPricer	External	✓	-
	deleteToken	External	✓	-
	withdraw	External	✓	-
	usdAmountToToken	Public		-
	enforceIsSupportedToken	External		-
Item	Implementation	IItem, ReentrancyGuardUpgradeable, UUPSUpgradeable, ERC721EnumerableUpgradeable, MulticallUpgradeable		

		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	mint	External	✓	nonReentrant
	burn	External	✓	-
	stopSell	External	✓	-
	setNewMaxSupply	External	✓	-
	setBaseUri	External	✓	-
	_baseURI	Internal		
FlexStakingStrategy	Implementation	IStakingStrategy, ReentrancyGuardUpgradeable, UUPSUpgradeable, MulticallUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	setEarnings	External	✓	-
	updateDeposits	Public	✓	-
	stake	External	✓	-
	claim	External	✓	-
	sell	External	✓	-
	stakingType	External		-
	currentPeriod	External		-

	timestampPeriod	Public		-
	claimTimestamp	External		-
	getAllExpiredMonths	Public		-
	estimateRewards	Public		-
	canSell	Public		-
	estimateSell	Public		-
	lastUpdatedEarningsPeriod	External		-
	_enforceIsStakedToken	Internal		
	_blockTimestamp	Internal		
FixStakingStrategy	Implementation	IStakingStrategy, ReentrancyGuardUpgradeable, UUPSUpgradeable, MulticallUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	stake	External	✓	-
	claim	External	✓	nonReentrant
	sell	External	✓	nonReentrant
	stakingType	External		-
	estimateRewards	Public		-
	canSell	Public		-
	estimateSell	Public		-

	claimTimestamp	External		-
	_enforceIsStakedToken	Internal		
	_blockTimestamp	Internal		
AddressBook	Implementation	UUPSUpgradeable, MulticallUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	setProductOwner	External	✓	-
	addItem	External	✓	-
	deleteItem	External	✓	-
	addStakingStrategy	External	✓	-
	deleteStakingStrategy	External	✓	-
	setTreasury	External	✓	-
	enforceIsProductOwner	Public		-
	enforceIsItemContract	External		-
	enforceIsStakingStrategyContract	External		-
Pricer	Implementation	IPricer, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		

	setCurrentPrice	External	✓	-
	decimals	External		-
	latestRoundData	External		-
DateTimeLib	Library			
	_daysFromDate	Internal		
	_daysToDate	Internal		
	timestampFromDate	Internal		
	timestampFromDateTime	Internal		
	timestampToDate	Internal		
	timestampToDateTime	Internal		
	isValidDate	Internal		
	isValidDateTime	Internal		
	isLeapYear	Internal		
	_isLeapYear	Internal		
	isWeekDay	Internal		
	isWeekEnd	Internal		
	getDaysInMonth	Internal		
	_getDaysInMonth	Internal		
	getDayOfWeek	Internal		
	getYear	Internal		
	getMonth	Internal		
	getDay	Internal		

	getHour	Internal		
	getMinute	Internal		
	getSecond	Internal		
	addYears	Internal		
	addMonths	Internal		
	addDays	Internal		
	addHours	Internal		
	addMinutes	Internal		
	addSeconds	Internal		
	subYears	Internal		
	subMonths	Internal		
	subDays	Internal		
	subHours	Internal		
	subMinutes	Internal		
	subSeconds	Internal		
	diffYears	Internal		
	diffMonths	Internal		
	diffDays	Internal		
	diffHours	Internal		
	diffMinutes	Internal		
	diffSeconds	Internal		
ITreasury	Interface			

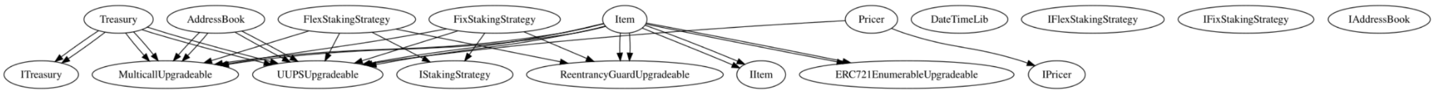
	enforceIsSupportedToken	External		-
	usdAmountToToken	External		-
	withdraw	External	✓	-
IStakingStrategy	Interface			
	stake	External	✓	-
IPricer	Interface			
	decimals	External		-
	latestRoundData	External		-
	setCurrentPrice	External	✓	-
IItem	Interface			
	burn	External	✓	-
	price	External		-
	maxSupply	External	✓	-
	totalMintedAmount	External	✓	-
	tokenStakingStrategy	External	✓	-
	mint	External	✓	-
	stopSell	External	✓	-
	setNewMaxSupply	External	✓	-
IFlexStakingStrategy	Interface			

	setEarnings	External	✓	-
	claim	External	✓	-
	sell	External	✓	-
IFixStakingStrategy	Interface			
	claim	External	✓	-
	sell	External	✓	-
IAddressBook	Interface			
	treasury	External		-
	enforceIsItemContract	External		-
	enforceIsProductOwner	External		-
	productOwner	External		-
	items	External		-
	stakingStrategies	External		-
	enforceIsStakingStrategyContract	External		-
Treasury	Implementation	ITreasury, UUPSUpgradeable, MulticallUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	setOnlyProductOwnerWithdrawn	External	✓	-

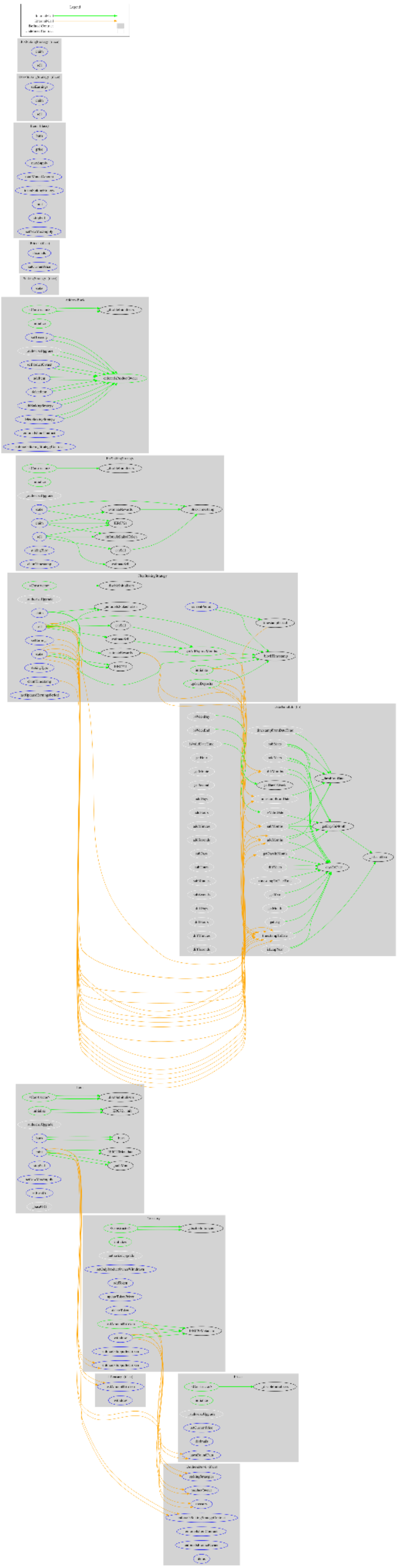
	addToken	External	✓	-
	updateTokenPricer	External	✓	-
	deleteToken	External	✓	-
	withdraw	External	✓	-
	usdAmountToToken	Public		-
	enforceIsSupportedToken	External		-
Item	Implementation	IItem, ReentrancyGuardUpgradeable, UUPSUpgradeable, ERC721EnumerableUpgradeable, MulticallUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	mint	External	✓	nonReentrant
	burn	External	✓	-
	stopSell	External	✓	-
	setNewMaxSupply	External	✓	-
	setBaseUri	External	✓	-
	_baseURI	Internal		
AddressBook	Implementation	UUPSUpgradeable, MulticallUpgradeable		

		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal		
	setProductOwner	External	✓	-
	addItem	External	✓	-
	deleteItem	External	✓	-
	addStakingStrategy	External	✓	-
	deleteStakingStrategy	External	✓	-
	setTreasury	External	✓	-
	enforceIsProductOwner	Public		-
	enforceIsItemContract	External		-
	enforceIsStakingStrategyContract	External		-

Inheritance Graph



Flow Graph



Summary

ElectraProject contract implements a NFT and staking mechanism. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io