Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2026.03.30, the SlowMist security team received the Aurevia Investment Consulting Limited team's security audit application for Atlas RWA, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.

Level	Description
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit

Serial Number	Audit Class	Audit Subclass
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

The Atlas RWA project is a comprehensive RWA smart contract platform that includes asset compliance control, primary market subscription, secondary market order book trading, rental distribution, and the platform token (ATK) economic mechanism (staking, fee deduction, and burning).

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Hardcoded Scaler in ATK Fee Calculation Assumes 6-Decimal Stablecoins	Arithmetic Accuracy Deviation Vulnerability	Critical	Fixed
N2	Insecure Chainlink Oracle Integration Enables Staleness and Unsafe Negative Casting	Design Logic Audit	High	Fixed
N3	Complete Absence of Slippage Protection Against ATK Price Volatility Causes Transaction Reversals	Design Logic Audit	Low	Fixed
N4	Decimal Conversion Truncation Leads To Capital Seizure Without Property Token Issuance	Arithmetic Accuracy Deviation Vulnerability	Low	Fixed
N5	Minimum Threshold Evasion	Others	Low	Fixed
N6	Dynamic Admin Fee Modification Provokes Arithmetic Underflow	Design Logic Audit	Medium	Fixed

NO	Title	Category	Level	Status
	DoS on Historical Engine Matches			
N7	Constrained 1e18 Decimal Arithmetic Permanently Sabotages Multi-Decimals Property RWA Listings	Arithmetic Accuracy Deviation Vulnerability	Suggestion	Acknowledged
N8	Order Query Unbounded Global Mapping Iteration Accelerates Out of Gas Collapses	Denial of Service Vulnerability	Suggestion	Acknowledged
N9	Defective Mempool Collision Logic Nullifies Sweep Interactions via Atomic Reverts	Design Logic Audit	Medium	Fixed
N10	Redundant Fee Splitting Operations Demonstrating Codebase Integration Misalignment	Others	Information	Fixed
N11	Suboptimal API Re-invention Bypassing Built-in Discount Mechanism	Others	Information	Fixed
N12	`discountBps` Semantic May Misinterpretation Induces Value Deflation	Others	Information	Acknowledged
N13	Last Recipient in Rent Distribution Receives All Accumulated Rounding Remainders	Design Logic Audit	Low	Fixed
N14	Gas DoS Paralysis Stemming from	Denial of Service Vulnerability	High	Fixed

NO	Title	Category	Level	Status
	Unbounded Linear Array Iterations			
N15	Universal Push-Protocol Transfer Rejection	Others	Information	Acknowledged
N16	External Call Reminder	Unsafe External Call Audit	Information	Acknowledged
N17	Incomplete Boundary Thresholds Entrap Isolated Residual Token Assets	Others	Suggestion	Acknowledged
N18	Risk of Excessive Privilege	Authority Control Vulnerability Audit	Medium	Acknowledged
N19	Token Compatibility Reminder	Others	Information	Acknowledged
N20	Arbitrary Dividend Accumulation Vulnerability via Unverified `amount` Parameters	Design Logic Audit	Medium	Fixed
N21	Rent Token Substitution Corrupts Monolithic Dividend Accumulators	Design Logic Audit	High	Fixed
N22	Omission of Regulatory Compliance Verifications Imposed Upon Property Sellers	Others	Suggestion	Fixed
N23	Unlimited Operation Lifespans Triggering Zombie Order Accumulation	Design Logic Audit	Suggestion	Acknowledged
N24	Unverified `recipients` Payload	Design Logic Audit	Medium	Fixed

NO	Title	Category	Level	Status
	Enables Arbitrary Rent Expropriation			
N25	Identical Mapping of Token Name and Symbol	Others	Information	Fixed
N26	Unrestricted `increaseAmount` Operation Neutralizes Time-Lock Constraints Enabling Flash Loan Exploits	Design Logic Audit	Medium	Fixed
N27	Arbitrary Timestamp Modifications Bypass Standardized Duration Configurations within `extendLock`	Design Logic Audit	Medium	Fixed
N28	Stale Position Exploitation Grants Perpetual Tier Privileges	Others	Information	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

File name and hash(SHA256):

contracts.zip: 4a9cb0caf8d1ee017a863959dbbb557abd65740f8aee1a5c326c4bbdfaaee24e

Fixed Version:

File name and hash(SHA256):

contracts.zip: a2d8e3284cda1243db21df3075a0b9b36f27d8854ff8040747e451a57f6d2c30

Audit Scope:

```
.
├── Compliance.sol
├── Exchange.sol
├── LockingVault.sol
├── PlatformToken.sol
├── PropertyToken.sol
├── Subscription.sol
└── interfaces
    └── IDividendReceiver.sol
```

The main network address of the contract is as follows:

Contract Address (BSC)	
Contract Name	Contract Address
Compliance (UUPS)	0x7aDdD777A3a1EE8C97F4392CeAA8689C0801D1F0 (Proxy) 0xB5aAa01E17C93633F48bD42Da794fCD06d13435D (Implementation)
PlatformToken (ATK)	0x4f129eF65C14Acd09160e354438145Fc4E13d88a
LockingVault (UUPS)	0x3162B7076d2384278F1A4BC7aA266492Cd45C4B3 (Proxy) 0x3688C341d6338b0900FC3C98eB9fD54DeB3345C0 (Implementation)
Exchange (UUPS)	0x963e882f75fF631c675148CB7902E1Fa18192165 (Proxy) 0x388973A288C8D1d415c0c6383e0B35a3bDF326b5 (Implementation)
PropertyToken BKK-02	0x9B033Ce0dBa35f1115A880BD55BCaF2C93589ae1
Subscription BKK-02 (UUPS)	0xc52798118BFe1b32d8d0dEf2A5e8d93d7Bd148D5 (Proxy) 0x3bb207bDF43965c5fDa54be305c52E179B472871 (Implementation)

Role Address		
Role Name	Address	Address Type
Platform Deployer	0x0E5ab990C31568649929FC78d317c14AC24f84f1	Safeheron MPC
BKK-02 Deployer	0x1496b52aEed7a8FCB2ab3e385F5d83Bd487570C6	Safeheron MPC

Role Address		
AutoSign	0x113db9fe5CFd6342025f1A46edC7830Cf5973a8c	Safeheron MPC
Whitelist Wallet	0xc34fFB571B75DC266006b8E362107a364604Caf8	Safeheron MPC
Relayer	0x42Ae6485f00d45Fff7F019d95c40141c8ad63953	Safeheron MPC

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

Compliance			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
setInvestorData	External	Can Modify State	onlyRole
batchSetInvestorData	External	Can Modify State	onlyRole
setFrozen	External	Can Modify State	onlyRole
setJurisdictionAllowed	External	Can Modify State	onlyRole
setAllowedJurisdictions	External	Can Modify State	onlyRole
isCompliant	External	-	-
isFrozen	External	-	-
isKYCPassed	External	-	-
getInvestor	External	-	-
isJurisdictionAllowed	External	-	-

Compliance			
_authorizeUpgrade	Internal	Can Modify State	onlyRole

Exchange			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole
createOrder	External	Can Modify State	nonReentrant whenNotPaused
cancelOrder	External	Can Modify State	nonReentrant
createBuyOrder	External	Can Modify State	nonReentrant whenNotPaused
cancelBuyOrder	External	Can Modify State	nonReentrant
matchOrders	External	Can Modify State	onlyRole nonReentrant whenNotPaused
buyOrder	External	Can Modify State	nonReentrant whenNotPaused
batchFillOrders	External	Can Modify State	nonReentrant whenNotPaused
_fillOrder	Internal	Can Modify State	-
getOrder	External	-	-
getOpenOrders	External	-	-

Exchange			
getOpenOrdersByProperty	External	-	-
setPropertyAllowed	External	Can Modify State	onlyRole
setStablecoinAllowed	External	Can Modify State	onlyRole
setUsdFeeRateBps	External	Can Modify State	onlyRole
setTreasury	External	Can Modify State	onlyRole
_calculateATKFee	Internal	-	-
_isPropertyAllowed	Internal	-	-
onDividendReceived	External	Can Modify State	nonReentrant
_settleDividend	Internal	Can Modify State	-
sweepDust	External	Can Modify State	onlyRole
_authorizeUpgrade	Internal	Can Modify State	onlyRole

PlatformToken			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20
setTreasury	External	Can Modify State	onlyRole
setPriceFeed	External	Can Modify State	onlyRole
setBurnBps	External	Can Modify State	onlyRole
burn	External	Can Modify State	-

PlatformToken			
previewATKFee	Public	-	-
processATKFee	External	Can Modify State	onlyRole
getATKPriceUSD	External	-	-
_burnWithFloor	Internal	Can Modify State	-

LockingVault			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
lock	External	Can Modify State	whenNotPaused nonReentrant
increaseAmount	External	Can Modify State	whenNotPaused nonReentrant
extendLock	External	Can Modify State	whenNotPaused
unlock	External	Can Modify State	nonReentrant
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole
getLockPosition	External	-	-
getUserTier	External	-	-
_isValidDuration	Internal	-	-
_authorizeUpgrade	Internal	Can Modify State	onlyRole

PropertyToken			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20
setStatus	External	Can Modify State	onlyRole
addToWhitelist	Public	Can Modify State	onlyRole
batchAddToWhitelist	External	Can Modify State	onlyRole
removeFromWhitelist	External	Can Modify State	onlyRole
setExchange	External	Can Modify State	onlyRole
burn	External	Can Modify State	onlyRole
mint	External	Can Modify State	onlyRole whenNotPaused
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole
depositRentAndDistribute	External	Can Modify State	onlyRole nonReentrant
distributeRentBatch	External	Can Modify State	onlyRole nonReentrant
getPropertyInfo	External	-	-
maxSupply	External	-	-
minTradeUnit	External	-	-
status	External	-	-
getHolders	External	-	-
_update	Internal	Can Modify State	whenNotPaused
_addHolder	Internal	Can Modify State	-

PropertyToken			
_removeHolder	Internal	Can Modify State	-
_distributeRent	Internal	Can Modify State	-
_buildTokenName	Private	-	-
_formatSequence	Private	-	-
_toString	Private	-	-

Subscription			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
setStablecoinAllowed	External	Can Modify State	onlyRole
setTreasury	External	Can Modify State	onlyRole
setPlatformToken	External	Can Modify State	onlyRole
setStablecoinFeeRateBps	External	Can Modify State	onlyRole
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole
_calculateFee	Internal	-	-
previewSubscription	Public	-	-
convertDecimals	Public	-	-
subscribe	External	Can Modify State	whenNotPaused nonReentrant
_authorizeUpgrade	Internal	Can Modify State	onlyRole

Subscription			

4.3 Vulnerability Summary

[N1] [Critical] Hardcoded Scaler in ATK Fee Calculation Assumes 6-Decimal Stablecoins

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In both the Subscription (`_calculateFee`) and Exchange (`_calculateATKFee`) contracts, mathematical procedures calculating platform ATK fees actively utilize a rigid `10**20` multiplier heavily assuming inputs maintain exactly 6 digits of precision (typical for USDC/USDT in the ETH main Chain). Specifically within `Subscription.sol`, the primary internal `_calculateFee(uint256 amount, bool payWithATK)` function receives the raw external parameter `amount` passed unaltered directly from the public-facing `subscribe()` and `subscribeWithPermit()` entry points. This `amount` natively carries the exact decimal configuration of the chosen external stablecoin. If universally compliant 18-decimal stablecoins (like DAI or USDT/USDC in the BSC Chain) are deposited, this static multiplication generates a significant computation error requesting trillions of unbacked ATK tokens natively breaking platform equivalency. While the codebase integrates a `convertDecimals` utility designed to normalize these dimensional mismatches, its usage is inconsistent; it is omitted during these specific fee calculations, resulting in unhandled dimensional misalignment.

Code location:

Subscription.sol#L158

Exchange.sol#L493

```
    function _calculateFee(
        uint256 amount,
        bool payWithATK
    ) internal view returns (uint256 feeInUSD, uint256 feeInToken) {
        if (payWithATK) {
            // ATK payment: 90% of stablecoin fee rate (10% ATK discount -- confirmed
```

```
rate)
            uint16 atkFeeRateBps = uint16((uint256(stablecoinFeeRateBps) * 9000) /
BPS_DENOMINATOR);
            feeInUSD = (amount * atkFeeRateBps) / BPS_DENOMINATOR;

            // Convert to ATK amount using Chainlink price
            int256 atkPrice = PlatformToken(platformToken).getATKPriceUSD();
            // Chainlink price is 8 decimals, ATK is 18 decimals
            // feeInToken = feeInUSD * 1e20 / atkPrice
            feeInToken = (feeInUSD * 1e20) / uint256(atkPrice); // Subscription
        } else {
            ...
        }
    }

   function _calculateATKFee(uint256 usdFee) internal view returns (uint256
atkAmount, uint256 burned, uint256 toTreasury) {
        // usdFee: USDT amount (6 decimals)
        // atkPrice: USD price from Chainlink (8 decimals)
        // atkAmount: ATK tokens needed (18 decimals)
        // Formula: atkAmount = usdFee * 1e20 / atkPrice
        int256 atkPrice = PlatformToken(platformToken).getATKPriceUSD();
        atkAmount = (usdFee * 10**20) / uint256(atkPrice);      // Exchange

        burned = (atkAmount * 70) / 100;
        toTreasury = atkAmount - burned;
    }
```

Solution

It is recommended to refactor the current `convertDecimals` implementation to adhere strictly to standardized precision normalization logic. The `convertDecimals` implementation must incorporate upper-bound validation checks safeguarding against excessively large decimal input parameters (e.g., `require(decimalsFrom <= a certain range && decimalsTo <= a certain range)`) to prevent arithmetic overflow anomalies within the `10**decimals` executions. Furthermore, its invocation should be systematically relocated and enforced exactly prior to any fee scaling calculations (e.g., overriding the static `10**20` multiplier within `_calculateFee`), ensuring all inputs are dynamically standardized prior to protocol math execution.

Status

Fixed; The division operation functionally truncates the fee dust during ATK fee calculation, leading to negligible (< $0.000001) fee yield loss for the platform, which is considered economically safe.

[N2] [High] Insecure Chainlink Oracle Integration Enables Staleness and Unsafe Negative Casting

Category: Design Logic Audit

Content

In the PlatformToken contract, retrieving programmatic pricing values arbitrarily queries `latestRoundData()` isolating solely the execution integer without extracting or validating corresponding chronological freshness parameters (`updatedAt`). Consequently, heavily volatile environments precipitating network disconnections remain entirely undetected rendering staled matrices fundamentally valid. Furthermore, the missing boundaries cascade fundamentally into the Subscription and Exchange downstream architecture: the retrieved `int256` value is unsafely cast directly into an unsigned `uint256` divisor without preliminary boundary checking (`answer > 0`). If the unvalidated oracle structurally returns a negative flag (e.g.,-1), EVM Two's Complement processing mathematically converts this precisely into `type(uint256).max`, generating a gravitationally massive divisor that entirely collapses user fee costs strictly toward 0.

Code location:

PlatformToken.sol#L118-121

Subscription.sol#L115

Exchange.sol#L429

```
function getATKPriceUSD() external view returns (int256) {
     (, int256 answer, , , ) = IAggregatorV3(priceFeed).latestRoundData();
     return answer;
 }

    int256 atkPrice = PlatformToken(platformToken).getATKPriceUSD();
```

Solution

It is recommended to systematically establish global validation boundaries directly within `getATKPriceUSD()`

comprehensively asserting `require(answer > 0, "Invalid Price");`, `require(updatedAt > 0)`, alongside explicitly verifying chronological freshness guarantees executing `require(block.timestamp - updatedAt <= MAX_DELAY, "Stale Oracle");`.

Status

Fixed

[N3] [Low] Complete Absence of Slippage Protection Against ATK Price Volatility Causes Transaction Reversals

Category: Design Logic Audit

Content

In the Subscription and Exchange contracts, externally facing execution entries (`subscribe`, `buyOrder`) fail to request a maximum acceptable ATK slippage parameter (e.g., `uint256 maxATKFee`) from users electing to `payWithATK`. Because the protocol dynamically calculates the ATK fee equivalent at the precise moment of execution, transactions pending in the mempool are heavily exposed to market volatility. If the underlying ATK Token experiences abrupt downward price volatility, the oracle valuation adjusts downwards, mathematically requiring a larger volume of ATK tokens to satisfy the static USD fee requirement. Because the protocol strictly relies on users or routing frontends authorizing exact, calculated token limits for `transferFrom` operations prior to execution, tracking dynamic fees without slippage bounding means that natural price fluctuations will natively outpace the predefined allowance, universally resulting in insufficient allowance and triggering widespread transaction reverts (Denial of Service).

Code location:

Subscription.sol#L194-251

Exchange.sol#L344-417

```
    function subscribe(
        address stablecoin,
        uint256 amount,
        bool payWithATK
```

```
    ) external whenNotPaused nonReentrant {
        ...
        if (payWithATK) {
            IERC20 atkToken = IERC20(platformToken);
            uint256 atkAllowance = atkToken.allowance(msg.sender, address(this));
            if (atkAllowance < feeInToken) {
                revert InsufficientAllowance(atkAllowance, feeInToken);
            }
            atkToken.safeTransferFrom(msg.sender, address(this), feeInToken);
            atkToken.forceApprove(platformToken, feeInToken);
            PlatformToken(platformToken).processATKFee(feeInToken, 10000);
        } else {
            ...
        }
        ...
    }

    function _fillOrder(
        uint256 orderId,
        uint256 amount,
        address buyer,
        bool payWithATK
    ) internal {
        ...
        if (payWithATK) {
            ...
            (uint256 atkAmount, , ) = _calculateATKFee(discountedFeeInUSD);
            IERC20(platformToken).safeTransferFrom(buyer, address(this), atkAmount);
            IERC20(platformToken).forceApprove(platformToken, atkAmount);
            PlatformToken(platformToken).processATKFee(atkAmount, 10000);
            IERC20(order.stablecoin).safeTransferFrom(buyer, order.seller, totalCost);
        } else {
            ...
        }
        ...
    }
```

Solution

It is recommended to supplement the function signature interfaces with a `uint256 maxAtkSlippageLimit` variable and strictly guarantee `require(feeInToken <= maxAtkSlippageLimit)`.

Status

Fixed; Partially Fixed

[N4] [Low] Decimal Conversion Truncation Leads To Capital Seizure Without Property Token Issuance

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the Subscription contract, scaling higher decimal stablecoin amounts to lower-decimal Property Tokens invokes division that is inherently subject to EVM downward truncation. Crucially, the protocol subsequently charges the user's external wallet based on the initial untruncated `amount`, despite distributing shares based on the truncated integer payload. In excessive cases where `mintAmount == 0`, users deposit stablecoin sums entirely into the treasury while silently receiving no platform property token equity.

Code location:

Subscription.sol#L205-209

```
        uint256 mintAmount = convertDecimals(
             amount,
             IERC20Metadata(stablecoin).decimals(),
             IPropertyToken(propertyToken).decimals()
         );
         ...
         IERC20(stablecoin).safeTransferFrom(msg.sender, treasury, amount);
```

Solution

It is recommended to enforce `require(mintAmount > 0)`. Furthermore, instead of charging the original raw `amount`, the protocol should deliberately re-invoke the existing `convertDecimals` function to calculate the precise stablecoin cost for the finalized shares , ensuring users only pay for exactly what they structurally acquire.

Status

Fixed; Partially Fixed

[N5] [Low] Minimum Threshold Evasion

Category: Others

Content

In the Subscription contract, establishing proportional fees through basic mathematical division allows users interacting with minute dust quantities to subvert mathematical integer division, returning zero values (`amount * bps / 10000 = 0`). This permits highly active micro-transaction scraping to secure platform functionalities devoid of contributing to the protocol's fee accumulation.

Code location:

Subscription.sol#L145-164

```
    function _calculateFee(
        uint256 amount,
        bool payWithATK
    ) internal view returns (uint256 feeInUSD, uint256 feeInToken) {
        if (payWithATK) {
            ...
            feeInUSD = (amount * atkFeeRateBps) / BPS_DENOMINATOR;

            ...
        } else {
            ...
        }
    }
```

Solution

It is recommended to enforce a universal foundational execution charge boundary (e.g., `require(feeInUSD >= MIN_LIMIT)`) protecting administrative viability.

Status

Fixed

[N6] [Medium] Dynamic Admin Fee Modification Provokes Arithmetic Underflow DoS on Historical Engine Matches

Category: Design Logic Audit

Content

In the Exchange contract, buyer commitments isolate locked margin capitals (`lockedStablecoin`) strictly proportional to the platform fee prevalent upon initiation. If protocol owners dynamically adjust `usdFeeRateBps` upwards concurrently, subsequent asynchronous `matchOrders` resolving delayed settlement evaluates required deductions based on the newer, globally enhanced inflated fee configuration. This silently eclipses the user's available locked budget causing catastrophic mathematical underflow reversions paralyzing historical functionality en masse.

Code location:

Exchange.sol#L287-300

```
    function matchOrders(
         uint256 sellOrderId,
         uint256 buyOrderId,
         uint256 amount,
         bool useSellerPrice
     ) external onlyRole(MATCHER_ROLE) nonReentrant whenNotPaused {
         ...
         {
             uint256 settlePricePerToken = useSellerPrice ? sellOrder.pricePerToken :
buyOrder_.pricePerToken;
             uint8 dec = IERC20Metadata(sellOrder.stablecoin).decimals();
             totalCost = (amount * settlePricePerToken * (10 ** dec)) / (1e18 * 1e6);
             fee = (totalCost * usdFeeRateBps) / 10000;
         }
         ...
         buyOrder_.amount -= amount;
         buyOrder_.lockedStablecoin -= (totalCost + fee);
         ...
     }
```

Solution

It is recommended to actively memorize immutable configuration snapshots securely inside the user's localized `BuyOrder` struct mapping preserving temporal integrity.

Status

Fixed

[N7] [Suggestion] Constrained 1e18 Decimal Arithmetic Permanently Sabotages Multi-Decimals Property RWA Listings

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the Exchange contract, the computational architecture calculating secondary fractional prices generically integrates any configured `address propertyToken`, but strictly scales mathematics using a universally hardcoded `1e18` denominator. While the platform's native `PropertyToken` implementation correctly defaults to 18 decimal representations (presenting no immediate operational hazard), the `Exchange` completely neglects to validate whether whitelisted properties genuinely adhere to this 18-decimal architectural expectation. If administrators eventually whitelist specialized third-party real-world assets (RWA) established natively on lower-precision boundaries (e.g., 6-decimal or 0-decimal non-fungible equivalents), the hardcoded `1e18` division will systematically over-truncate equations, reducing target `totalCost` resolutions downward exponentially to `0`.

Code location:

Exchange.sol#L212-368

```
        uint256 totalCost = (amount * order.pricePerToken * (10 ** stablecoinDecimals))
/ (1e18 * 1e6);
```

Solution

It is suggested to either actively validate precision configurations during whitelist integrations (`require(IERC20Metadata(propertyToken).decimals() == 18, "Unsupported Precision");`), or eliminate static assumptions altogether in favor of dynamically querying `10 ** IERC20Metadata(propertyToken).decimals()`.

Status

Acknowledged; After communicating with the project team, they stated that all PropertyTokens are fixed at 18 decimal places, and the contract does not support third-party assets.

[N8] [Suggestion] Order Query Unbounded Global Mapping Iteration Accelerates Out of Gas Collapses

Category: Denial of Service Vulnerability

Content

In the Exchange contract, frontend enumeration view sequences explicitly parse the unconstrained exhaustive entirety of structural active orders unconditionally processing through immense localized index chains mapping array sizes. Continuous high-volume growth definitively promises encountering the EVM Block Gas Limit eventually eliminating all exterior protocol state viewing properties via Out of Gas exceptions.

Code location:

Exchange.sol#L441-460

```
    function getOpenOrdersByProperty(address propertyToken) external view returns
(uint256[] memory) {
        uint256[] storage propertyOrderList = _propertyOrders[propertyToken];
        uint256 count = 0;

        for (uint256 i = 0; i < propertyOrderList.length; i++) {
            if (orders[propertyOrderList[i]].active) {
                count++;
            }
        }

        uint256[] memory result = new uint256[](count);
        uint256 index = 0;
        for (uint256 i = 0; i < propertyOrderList.length; i++) {
            uint256 orderId = propertyOrderList[i];
            if (orders[orderId].active) {
                result[index++] = orderId;
            }
        }
        return result;
    }
```

Solution

It is recommended to decouple active state representations off-chain entirely (via The Graph indexers), or alternatively format explicit `offset` and `limit` boundaries establishing pagination indexing structurally across analytical arrays.

Status

Acknowledged; After communicating with the project team, they stated that the front-end has already queried through the back-end indexer; the on-chain view is only for backup.

[N9] [Medium] Defective Mempool Collision Logic Nullifies Sweep Interactions via Atomic Reverts

Category: Design Logic Audit

Content

In the Exchange contract, Taker executions mandate purely exact volumetric thresholds. Given concurrent mempool ordering volatility, microscopic external interactions inadvertently extracting dust increments out of active targeted structures silently alters expected mathematical remainder states. Consequently, standard bulk-sweeper routines (`batchFillOrders`) suffer from unrecoverable atomic execution reverts merely due to marginal isolated miscalculations resulting in completely deteriorated user interface acquisition capabilities.

Code location:

Exchange.sol#L352

```
        if (order.amount < amount) revert InsufficientOrderAmount(order.amount,
amount);
```

Solution

It is recommended to restructure strict evaluation barriers into flexible downgrading boundaries securing adaptive completion metrics.

Status

Fixed; Enhancement Recommendation: Because the `maxATKFee` check in the `_fillOrder` function is performed

independently within the loop body, an error triggered by a single order exceeding the slippage limit will again block the entire bulk buy transaction (Atomic Revert).

Recommendation: In `batchFillOrders`, if `atkAmount > maxATKFee` is encountered in a single loop, a graceful skip of the abnormal order should be implemented using `continue` instead of throwing a crash (Revert); alternatively, the anti-slippage check could be moved outside the loop body, accumulating all successfully scanned `TotalATKAccumulated` values and comparing them to the unique `Global_maxATKFee` at the end of the function.

[N10] [Information] Redundant Fee Splitting Operations Demonstrating Codebase Integration Misalignment

Category: Others

Content

In the Exchange contract, the internal `_calculateATKFee()` function explicitly computes three return values: `atkAmount`, `burned` (70% of the total), and `toTreasury` (the remaining 30%). However, at the actual call site within `_fillOrder()`, the caller deliberately discards both `burned` and `toTreasury` via empty tuple assignment (`(uint256 atkAmount, , )`), retaining only the total `atkAmount`. The discarded values are never utilized because the actual burn-and-treasury splitting logic is independently and redundantly re-performed inside `PlatformToken.processATKFee()`, which internally invokes its own `previewATKFee()` to recalculate the identical proportional distribution from scratch. This indicates a codebase integration misalignment where `Exchange._calculateATKFee` duplicates splitting logic that is already authoritatively handled by `PlatformToken`.

Code location:

Exchange.sol#L382, L487-497

```
    function _calculateATKFee(uint256 usdFee) internal view returns (uint256
atkAmount, uint256 burned, uint256 toTreasury) {
        // usdFee: USDT amount (6 decimals)
        // atkPrice: USD price from Chainlink (8 decimals)
        // atkAmount: ATK tokens needed (18 decimals)
```

```
        // Formula: atkAmount = usdFee * 1e20 / atkPrice
        int256 atkPrice = PlatformToken(platformToken).getATKPriceUSD();
        atkAmount = (usdFee * 10**20) / uint256(atkPrice);

        burned = (atkAmount * 70) / 100;
        toTreasury = atkAmount - burned;
    }
```

Solution

It is recommended to simplify `_calculateATKFee` to return only the `atkAmount` value that is actually consumed, removing the redundant `burned` and `toTreasury` computations to eliminate dead code and reduce gas overhead.

Status

Fixed

[N11] [Information] Suboptimal API Re-invention Bypassing Built-in Discount Mechanism

Category: Others

Content

In the PlatformToken contract, the `processATKFee(uint256 baseFeeAmount, uint16 discountBps)` function was explicitly designed with a built-in `discountBps` parameter to handle discount calculations internally via `previewATKFee`: `chargedAmount = (baseFeeAmount * discountBps) / BPS_DENOMINATOR`. However, both the Exchange and Subscription contracts completely bypass this built-in discount mechanism. In `Exchange._fillOrder()`, the 10% ATK discount is manually computed externally (`discountedFeeInUSD = (feeInUSD * 9000) / 10000`), and the already-discounted result is then fed into `_calculateATKFee` to derive the ATK amount. Subsequently, `processATKFee` is invoked with a hardcoded `10000` (meaning "no further discount"), effectively nullifying its own discount parameter. The `Subscription` contract exhibits the identical pattern, also passing a hardcoded `10000` to `processATKFee`. This duplication of discount logic outside of the API creates maintenance risk: if the discount rate changes, developers must remember to update it in multiple locations rather than a single source.

Code location:

Exchange.sol#L380-391

Subscription.sol#L242

```
uint256 discountedFeeInUSD = (feeInUSD * 9000) / 10000;
 (uint256 atkAmount, , ) = _calculateATKFee(discountedFeeInUSD);
 ...
 PlatformToken(platformToken).processATKFee(atkAmount, 10000);

 PlatformToken(platformToken).processATKFee(feeInToken, 10000);
```

Solution

It is recommended to consolidate discount logic by passing the raw (undiscounted) fee amount and the actual `discountBps` value (e.g., `9000`) directly into `processATKFee`, allowing the `PlatformToken` to serve as the single source for discount application.

Status

Fixed; Optimization suggestion: The current asynchronous refund logic causes users to lock an additional 10% of their ATK tokens until the order is matched, wasting the use of users' ATK tokens. It is recommended to directly use `previewATKFee` to calculate the 9000 discount, instead of using the full amount collection and refund method.

[N12] [Information] `discountBps` Semantic May Misinterpretation Induces Value Deflation

Category: Others

Content

In the PlatformToken contract, administrative configuration arguments logically identifying percentage `discountBps` values mathematically apply these configuration vectors directly as overarching Absolute Charge Multipliers completely reversing expected fiscal deduction models (`charge = base * discount`). Initiating explicit 10% reductions (1000 bps) mathematically obligates users issuing exclusively 10% underlying costs completely destroying 90% of targeted protocol baseline revenue targets.

Code location:

PlatformToken.sol#L78-93

```
    function previewATKFee(
        uint256 baseFeeAmount,
        uint16 discountBps
    ) public view returns (uint256 chargedAmount, uint256 burnAmount, uint256
treasuryAmount) {
        if (discountBps > BPS_DENOMINATOR) revert InvalidDiscountBps();
        chargedAmount = (baseFeeAmount * discountBps) / BPS_DENOMINATOR;
        ...
    }
```

Solution

It is recommended to rename the `discountBps` parameter as `chargeBps`.

Status

Acknowledged

[N13] [Low] Last Recipient in Rent Distribution Receives All Accumulated Rounding Remainders

Category: Design Logic Audit

Content

In the PropertyToke contract's `_distributeRent()` function, each recipient's share is calculated proportionally via `(amount * balanceOf(recipient)) / supply`, except for the last recipient in the array, who unconditionally receives `amount - distributed` (the entire unallocated remainder). This design was intended to handle rounding dust, but it creates a flaw: if the `recipients` array provided by the `RENT_ADMIN_ROLE` does not cover all holders, the last recipient absorbs the aggregate undistributed pool — potentially receiving a disproportionately large share. Additionally, the magnitude of the remainder is directly influenced by the order and composition of the `recipients` array, meaning the admin can manipulate who receives the surplus simply by controlling the array ordering.

Code location:

PropertyToken.sol#L253-255

```
            if (i == recipients.length - 1) {
                share = amount - distributed;
            } else {
```

```
                share = (amount * balanceOf(recipient)) / supply;
                distributed += share;
            }
```

Solution

It is recommended to calculate all recipients' shares proportionally without exception, and handle any remaining dust by either retaining it in the contract for future distribution or transferring it to the treasury.

Status

Fixed

[N14] [High] Gas DoS Paralysis Stemming from Unbounded Linear Array Iterations

Category: Denial of Service Vulnerability

Content

In the PropertyToken contract, architectural design relies on unbounded global arrays (`_holders`) processed comprehensively during structural modifications natively. Specifically, `_removeHolder()` dynamically discovers individual terminating addresses invoking expansive `for` loop iterations processing expanding index databases unyieldingly. As the holder volume organically explodes into the thousands, executing these procedures unequivocally guarantees surpassing the hardcoded EVM Block Gas Limit constraints. This flaw creates a highly toxic synergy with the `Exchange` contract's `batchFillOrders()` function: processing a batch of orders triggers successive token transfers, each dynamically invoking `_removeHolder()` internally. This compounding O(N) complexity quadratically inflates gas expenditures per batch, systematically guaranteeing immediate `Out of Gas` (OOG) crashes that irreversibly paralyze all macroscopic `Exchange` batch-filling workflows alongside crippling centralized rent manipulations (`depositRentAndDistribute()`).

Code location:

PropertyToken.sol#L200-242

Exchange.sol#L328-342

```
    function _update(address from, address to, uint256 value) internal override
whenNotPaused {
        ...
        if (from != address(0) && balanceOf(from) == 0) {
            _removeHolder(from);
        }
        if (to != address(0) && balanceOf(to) > 0) {
            _addHolder(to);
        }
    }

    function _addHolder(address account) internal {
        ...
    }

    function _removeHolder(address account) internal {
        if (!_isHolder[account]) return;

        _isHolder[account] = false;
        uint256 len = _holders.length;
        for (uint256 i = 0; i < len; ++i) {
            if (_holders[i] == account) {
                _holders[i] = _holders[len - 1];
                _holders.pop();
                emit HolderRemoved(account);
                break;
            }
        }
    }

   function batchFillOrders(
        uint256[] calldata orderIds,
        uint256[] calldata amounts,
        bool payWithATK
    ) external nonReentrant whenNotPaused returns (uint256 totalFilled) {
        ...
    }
```

Solution

It is recommended to completely discard basic array iteration indexing for storage maintenance: fundamentally

implement OpenZeppelin's `EnumerableSet.AddressSet`, or establish an O(1) swap-and-pop array deletion mechanism tracked via an auxiliary `mapping(address => uint256) holderIndex`.

Status

Fixed

[N15] [Information] Universal Push-Protocol Transfer Rejection

Category: Others

Content

In the PropertyToken contract, administrative distribution implementations linearly initiate sequential `safeTransfer` iterations actively forwarding dividends manually across all holders. Within this unyielding loop, if any single receiving address encounters a transfer failure—such as triggering a USDT/USDC centralized regulatory blacklist freeze, or directing funds towards any malicious/incompatible contract address inherently structured to `revert` upon receiving specific token transfers—the foundational `safeTransfer` collapses. This singular isolated failure immediately propagates backwards, entirely reverting the overarching algorithmic transaction and permanently obstructing all uncompromised subsequent holders from securing their legitimate rental distributions.

Code location:

PropertyToken.sol#L261

```
                rentToken.safeTransfer(recipient, share);
```

Solution

It is recommended to migrate from Push-based distribution to a Pull-based model where each holder calls `claimDividend()` to withdraw their own share. Alternatively, wrap each `safeTransfer` call inside a `try/catch` block so that a single failed transfer does not revert the entire distribution loop.

Status

Acknowledged

[N16] [Information] External Call Reminder

Category: Unsafe External Call Audit

Content

In the PropertyToken contract, the protocol forcefully executes a dynamic external callback hook `try IDividendReceiver(recipient).onDividendReceived(...)` when iterating through recipient loops during rent allocations. While wrapped safely utilizing `try/catch` guarding against direct operational revert DoS, delegating execution flow outwards natively before finalizing state accounting variables allows sophisticated malicious recipient contracts to enact dangerous Re-entrancy attacks or manipulate external routing components.

Code location:

PropertyToken.sol#L265-L271

```
                if (recipient.code.length > 0) {
                    try
IDividendReceiver(recipient).onDividendReceived(_propertyInfo.rentToken, share) {
                        // Hook called successfully
                    } catch {
                        // Not implementing interface or call failed, ignore
                    }
                }
```

Solution

N/A

Status

Acknowledged

[N17] [Suggestion] Incomplete Boundary Thresholds Entrap Isolated Residual Token Assets

Category: Others

Content

In the PropertyToken contract, validation procedures mandating external trading boundary minimum limits strictly

target only the transitional `value` metrics, explicitly ignoring the subsequent remnant quantities remaining anchored inside the originating account. Furthermore, the overarching `mint` and `burn` mechanics completely bypass `minTradeUnit` assertions entirely, allowing administrators to arbitrarily mint dust (e.g., 1 wei) directly toward isolated portfolios. Consequently, operational transactions marginally exceeding limits natively strand fractional terminal balances definitively below absolute extraction threshold limits natively, permanently seizing associated capital residues.

Code location:

PropertyToken.sol#L206

```
    function _update(address from, address to, uint256 value) internal override
whenNotPaused {
        if (from != address(0) && to != address(0)) {
            ...
            if (value < _propertyInfo.minTradeUnit) {
                revert BelowMinTradeUnit(value, _propertyInfo.minTradeUnit);
            }
        }
        ...
    }
```

Solution

It is recommended to enforce a minimum amount check on `mint` operations (`require(amount >= minTradeUnit)`), preventing administrators from creating sub-threshold dust positions at the source.

Status

Acknowledged

[N18] [Medium] Risk of Excessive Privilege

Category: Authority Control Vulnerability Audit

Content

In the Subscription, Exchange, PlatformToken, PropertyToken, and Compliance contracts, several administrative

roles (such as `DEFAULT_ADMIN_ROLE`, `UPGRADER_ROLE`, `COMPLIANCE_ADMIN_ROLE`, `MINTER_ROLE`, and `BURNER_ROLE`) hold excessive privileges, creating a significant centralization risk.

The administrators can continuously modify the `treasury` address holding protocol incomes, change the `usdFeeRateBps` and `atkFeeRateBps` fee structures, and alter critical configuration parameters.

The `COMPLIANCE_ADMIN_ROLE` can single-handedly freeze or unfreeze any investor's account, allowing a compromised admin to selectively freeze innocent users and lock users' funds in the platform. Furthermore, the `DEFAULT_ADMIN_ROLE` retains exclusive control over the overarching Access Control lifecycles, enabling them to grant or revoke any roles, replace the underlying logic implementation via UUPS upgrades leveraging `UPGRADER_ROLE`, and forcibly execute arbitrary algorithmic modifications.

A malicious or compromised `MINTER_ROLE` or `BURNER_ROLE` in the `PropertyToken` structure could arbitrarily mint infinite tokens to themselves or forcefully burn equity balances directly from investors' addresses, massively diluting capital or directly stealing wealth. The central administrator can also discriminatorily distribute massive aggregated rents via `depositRentAndDistribute` and `distributeRentBatch` to specific configurations.

Code location:

Subscription.sol#L109-134, L253

Exchange.sol#L257-318, L462-485, L553

Compliance.sol#L86-154, L212

PlatformToken.sol#L53-72

PropertyToken.sol#L94-145, L155-178

```
function setFeeRateBps
function setAtkFeeRateBps
function setTreasury
function setFrozen
function setInvestorData
function setAllowedJurisdictions
function mint
function burn
function depositRentAndDistribute
function distributeRentBatch
```

```
function grantRole
function _authorizeUpgrade
```

Solution

In the short term, transferring owner ownership to multisig contracts is an effective solution to avoid single-point risk. But in the long run, it is a more reasonable solution to implement a privilege separation strategy and set up multiple privileged roles to manage each privileged function separately. And the authority involving user funds should be managed by the community, and the EOA address can manage the authority involving emergency contract suspension. This ensures both a quick response to threats and the safety of user funds.

Status

Acknowledged; After communicating with the project team, they stated that all administrative operations are executed via a Safeheron MPC wallet (3 Key Shares, 2-of-3 signing threshold). The production Co-Signer EC2 instance is further protected by IP whitelisting and a Callback approval policy. Second, real estate tokenization operates under legal frameworks that mandate KYC enforcement, account freezing, rent distribution, and property status management. Third, during the current MVP phase, the platform serves a limited set of invited, KYC-verified qualified investors with constrained trading volumes and position sizes, where centralized management capability is a regulatory prerequisite for rapid incident response (e.g., account freezing upon regulatory request). Finally, the project team confirmed that their Phase 2 roadmap already includes the introduction of Timelock contracts (enforcing delayed execution with a public notice period for critical operations) and multi-signature governance upgrades, which will be progressively implemented as user scale and AUM grow. Current mitigation measures also include comprehensive audit logging for all administrative operations (with 365-day retention via CloudWatch) and restricting the UPGRADER_ROLE exclusively to the deployer address, with daily operational wallets holding no upgrade privileges.

Through on-chain contract verification, the project team has configured the corresponding role addresses as the corresponding contract addresses or deployer addresses. Both the deployer and the corresponding EOA role addresses are managed by the 2-of-3 Safeheron MPC wallet.

[N19] [Information] Token Compatibility Reminder

Category: Others

Content

The existing accounting models fundamentally assume that all interacting stablecoins and functional tokens firmly obey standard ERC-20 primitive implementations mapping 1-to-1 transfer integrities. Integrating non-standard overarching token specifications, particularly Fee-on-Transfer (FOT) tokens, Rebasing (Elastic Supply) base tokens, ERC-777 hook-enabled tokens, or heavily deflationary properties, will entirely decouple internal ledger evaluations resulting in unrecoverable systemic insolvency. For instance, if an elastic rebasing algorithmic token or a fee-deducting asset acts as the stablecoin, the mathematically recorded margin balances will massively diverge from the actual contractual holding reality, effectively trapping funds permanently due to accounting mismatches.

Code location:

Subscription.sol

Exchange.sol

PropertyToken.sol

```
funtion safeTransferFrom
funtion safeTransfer
```

Solution

It is recommended to explicitly document and mechanically whitelist exclusively rigid standard foundational ERC-20 structures (USDC/USDT standard implementations). Alternatively, implement dynamic pre-and-post `balanceOf` checkpoint comparisons calculating precise exacted extraction delta quantities isolating actual payload deliveries securely.

Status

Acknowledged; After communicating with the project team, they stated that only USDT/USDC (standard ERC-20) is supported; the contract does not intend to support FOT/Rebase tokens.

[N20] [Medium] Arbitrary Dividend Accumulation Vulnerability via Unverified `amount` Parameters

Category: Design Logic Audit

Content

In the Exchang contract, the `onDividendReceived()` callback passively trusts the `amount` parameter provided by external callers to update the global `dividendPerToken` accumulator. It fundamentally fails to cryptographically verify whether the `Exchange` actually received the declared `rentToken` volume. A malicious or compromised `PropertyToken` wrapper can invoke this callback explicitly feeding a massive counterfeit `amount`, artificially skyrocketing the accumulator. Consequently, whenever subsequent users trigger `_settleDividend()`, the protocol will attempt to transfer profoundly inflated rent rewards outwards. Because the actual contractual balance entirely lacks these phantom tokens, the underlying `safeTransfer` will consistently revert due to insufficient native balance resulting in a permanent Denial of Service (DoS) across all core order operations (canceling, matching, and executing limit orders).

Code location:

Exchange.sol#L504-L517

```
    function onDividendReceived(address rentToken, uint256 amount) external override
nonReentrant {
        address propertyToken = msg.sender;
        if (!allowedProperties[propertyToken]) revert
PropertyNotAllowed(propertyToken);

        propertyToRentToken[propertyToken] = rentToken; // Always update (H-3)

        uint256 exchangeBalance = IERC20(propertyToken).balanceOf(address(this));
        if (exchangeBalance > 0) {
            dividendPerToken[propertyToken] += (amount * 1e18) / exchangeBalance;
        }
        // exchangeBalance == 0: amount stays as dust (admin can sweep)

        emit DividendReceived(propertyToken, rentToken, amount);
    }
```

Solution

It is recommended to implement the exact balance delta calculation paradigm ensuring only physically received assets alter the accumulator. Additionally, certain access restrictions can be added to onDividendReceived.

Status

Fixed

[N21] [High] Rent Token Substitution Corrupts Monolithic Dividend Accumulators

Category: Design Logic Audit

Content

In the Exchange contract, the `propertyToRentToken[propertyToken] = rentToken` mapping designation can be dynamically overwritten during successive `onDividendReceived()` invocations. Alarmingly, the `dividendPerToken` accumulator acts as a monolithic, singular registry entirely stripped of `rentToken` dimension differentiation. If a property administrator changes the property's operational rent token structurally (e.g., from USDC to USDT), the historic fractional numerical accumulation remains permanently fossilized inside the monolithic registry. Subsequent users will claim dividends calculated combining both structural eras but entirely withdrawn against the newly designated `rentToken`'s specific balance. This mathematical cross-contamination guarantees systemic insolvency as the active pool mechanically fails to cover hybridized historical requisitions, perpetually reverting `_settleDividend` and locking active historical orders.

Code location:

Exchange.sol#L504-L517

```
    function onDividendReceived(address rentToken, uint256 amount) external override
nonReentrant {
        address propertyToken = msg.sender;
        if (!allowedProperties[propertyToken]) revert
PropertyNotAllowed(propertyToken);

        propertyToRentToken[propertyToken] = rentToken; // Always update (H-3)

        uint256 exchangeBalance = IERC20(propertyToken).balanceOf(address(this));
```

```
        if (exchangeBalance > 0) {
            dividendPerToken[propertyToken] += (amount * 1e18) / exchangeBalance;
        }
        // exchangeBalance == 0: amount stays as dust (admin can sweep)

        emit DividendReceived(propertyToken, rentToken, amount);
    }
```

Solution

It is recommended to maintain structurally detached accumulators mapping intricately to their independently respective `rentToken` dimensions, or explicitly prohibit and revert the functional substitution of a `rentToken` while property-specific limit orders persist.

Status

Fixed

[N22] [Suggestion] Omission of Regulatory Compliance Verifications Imposed Upon Property Sellers

Category: Others

Content

During the initialization of new listings via `createOrder()`, the Exchange contract firmly validates whether the overarching `PropertyToken` infrastructure is whitelisted via `_isPropertyAllowed()`. However, the architecture completely neglects verifying the active KYC organizational compliance or systemic freezing status associated with the `msg.sender` (the originating Seller). As a result, comprehensively sanctioned, frozen, or entirely non-compliant addresses preserve unimpeded capabilities to generate expansive sell orders, persistently polluting the structural active orderbook organically and initiating unregulated liquidity entanglements.

Code location:

Exchange.sol#L147-181

```
    function createOrder(
        address propertyToken,
        uint256 amount,
        uint256 pricePerToken,
```

```
        address stablecoin
    ) external nonReentrant whenNotPaused {
        ...
    }
```

Solution

It is recommended to actively integrate compliance validation barriers targeting the originator address executing identically tracking broader infrastructure protections.

Status

Fixed

[N23] [Suggestion] Unlimited Operation Lifespans Triggering Zombie Order Accumulation

Category: Design Logic Audit

Content

Order and BuyOrder blueprints natively structurally lack temporal expiration dimensions. Generated trading commitments consequently reside perpetually active inside iterative evaluation environments unless successfully matched procedurally or expressly cancelled manually by originators. Over prolonged operational cycles, forgotten, functionally abandoned, or organically bypassed dust-valued commitments progressively accumulate into "zombie orders," aggressively worsening the computational parsing load natively across mapping iterations, definitively provoking universal `Out of Gas` execution anomalies natively.

Code location:

Exchange.sol #L31-52

```
    struct Order {
        uint256 id;
        address seller;
        address propertyToken;
        uint256 amount;
        uint256 pricePerToken;
        address stablecoin;
        bool active;
        uint256 createdAt;
```

```
    }

    struct BuyOrder {
        uint256 id;
        address buyer;
        address propertyToken;
        uint256 amount;              // remaining PT amount to buy (18 decimals)
        uint256 pricePerToken;       // price per token (6 decimals)
        address stablecoin;
        uint256 lockedStablecoin;    // remaining locked USDT (stablecoin decimals)
        bool active;
        uint256 createdAt;
    }
```

Solution

It is recommended to proactively incorporate an explicit `uint256 expiresAt` Unix timestamp boundary inside conventional schema configurations, unconditionally asserting `require(block.timestamp < order.expiresAt)` within sweeping resolution evaluations dynamically.

Status

Acknowledged; After communicating with the project team, they stated that the backend relayer will manage expiration logic, but adding expiration on-chain increases gas costs and complexity.

[N24] [Medium] Unverified `recipients` Payload Enables Arbitrary Rent Expropriation

Category: Design Logic Audit

Content

In the PropertyToken contract, the `distributeRentBatch()` function accepts an entirely unverified `address[] calldata recipients` array provided by the `RENT_ADMIN_ROLE`. The function fails to validate that provided addresses are actual holders (`balanceOf(recipient) > 0`), omits protections against duplicate entries, and does not verify whether the array covers the complete `_holders` registry. While a non-holder address inserted at a non-terminal position would receive `share = (amount * 0) / supply = 0` (harmless), this vulnerability compounds critically with N18's last-index remainder logic: if a malicious `RENT_ADMIN_ROLE` deliberately places an attacker-

controlled zero-balance address as the final element of the `recipients` array, the last-index branch (`share = amount - distributed`) bypasses proportional calculation entirely, awarding the attacker the aggregate undistributed rent pool regardless of holding zero tokens.

Code location:

PropertyToken.sol#L166-178

```
function distributeRentBatch(
    uint256 amount,
    address[] calldata recipients
) external onlyRole(RENT_ADMIN_ROLE) nonReentrant {
    if (amount == 0 || recipients.length == 0) revert InvalidValue();
    if (totalSupply() == 0) revert InvalidValue();
    ...
    _distributeRent(amount, recipients);
    emit RentDistributedBatch(amount, recipients.length);
}
```

Solution

It is recommended to validate every recipient via `require(balanceOf(recipient) > 0)` during array iteration, implement duplicate address detection, or deprecate arbitrary input arrays entirely by restricting distribution to the internal authenticated `_holders` registry.

Status

Fixed

[N25] [Information] Identical Mapping of Token Name and Symbol

Category: Others

Content

In the PropertyToken contract, the ERC-20 constructor applies the same `_buildTokenName()` output to both the `name` and `symbol` parameters (`ERC20(_buildTokenName(...), _buildTokenName(...))`). Industry standards require abbreviated tickers for `symbol` (e.g., `aRE-NYC`) clearly differentiated from descriptive `name`

strings (e.g., `Atlas Real Estate - New York #01`). Using identical elongated strings for both breaks wallet and aggregator display conventions. Furthermore, within the `Exchange` contract, multiple `PropertyToken` deployments representing different properties can share identical `name` and `symbol` strings while having entirely different contract addresses. This creates significant user confusion risks on the trading interface, as buyers cannot visually distinguish between distinct property assets listed under the same token identifier, potentially leading to accidental purchases of unintended properties or enabling phishing-style token impersonation.

Code location:

PropertyToken.sol#L70

```
constructor(
        string memory cityCode_,
        uint256 sequence_,
        ...
    ) ERC20(_buildTokenName(cityCode_, sequence_), _buildTokenName(cityCode_,
sequence_)) {...}
```

Solution

It is recommended to implement a separate `_buildTokenSymbol()` function generating abbreviated, unique tickers for the `symbol` parameter. Additionally, consider incorporating a unique property identifier (e.g., property ID or address fragment) within the symbol to prevent naming collisions across different PropertyToken deployments listed on the Exchange.

Status

Fixed

[N26] [Medium] Unrestricted `increaseAmount` Operation Neutralizes Time-Lock Constraints Enabling Flash Loan Exploits

Category: Design Logic Audit

Content

In the LockingVault contract, while the primary `lock()` function correctly enforces a mandatory temporal locking

increment (`block.timestamp + duration`), the supplementary `increaseAmount()` function inherently fails to proportionally extend or recalculate the corresponding `unlockTime`. Furthermore, it fundamentally omits asserting whether the targeted existing position has already functionally expired. This architectural oversight may introduce a instantaneous Flash Loan attack vector: an attacker can establish an empty "shell" position containing negligible dust and patiently map it until temporal expiration. Subsequently, during high-value governance distributions or snapshot events, they identically utilize Flash Loans within a solitary transaction block to artificially inject millions of tokens via `increaseAmount()` (instantly securing maximum organizational Tier credentials), cleanly executing reward claims, and immediately executing `unlock()` stripping all assets uniformly returning loans devoid of penalties definitively.

Code location:

LockingVault.sol#L91-L103

```
function increaseAmount(uint256 amount) external whenNotPaused nonReentrant {
     if (amount == 0) revert InvalidAmount();
     LockPosition storage position = _positions[msg.sender];
     if (position.amount == 0) revert NoPosition();
     position.amount += amount;
     totalLocked += amount;
     atkToken.safeTransferFrom(msg.sender, address(this), amount);
     emit AmountIncreased(msg.sender, amount, position.amount);
 }
```

Solution

It is strictly recommended to rigorously mandate time-locking calculations during supplementary liquidity injections. Alternatively, implement dynamic proportional extensions explicitly resetting baseline configurations.

Status

Fixed

[N27] [Medium] Arbitrary Timestamp Modifications Bypass Standardized Duration Configurations within `extendLock`

Category: Design Logic Audit

Content

Within the LockingVault initialization phase (`lock()`), administrators meticulously force structural participation isolating standardized operational duration blocks explicitly (`30` , `60` , `90` , `180` days) via rigid programmatic bounds (`_isValidDuration()`). Conversely, the `extendLock()` continuation mechanics entirely dismantle these precise mathematical constraints, singularly accepting unregulated primitive absolute timestamp integers uniformly executing isolated bounds evaluating strictly `if (newUnlockTime <= position.unlockTime) revert;` . This incongruity actively empowers manipulative participants automating microscopic extensions (e.g., adding `1 second`) effectively freezing maximum active Tier participations bypassing standardized 30-day epoch participation requisitions actively.

Code location:

LockingVault.sol#L105-114

```
function extendLock(uint256 newUnlockTime) external whenNotPaused {
     LockPosition storage position = _positions[msg.sender];
     if (position.amount == 0) revert NoPosition();
     if (newUnlockTime <= position.unlockTime) revert CannotShortenLock();

     uint256 previousUnlockTime = position.unlockTime;
     position.unlockTime = newUnlockTime;

     emit LockExtended(msg.sender, previousUnlockTime, newUnlockTime);
 }
```

Solution

It is recommended to systematically deprecate receiving unregulated arbitrary absolute Unix timestamps, strictly returning architectures back towards standard epoch duration increments injecting natively bounded arguments.

Status

Fixed

[N28] [Information] Stale Position Exploitation Grants Perpetual Tier Privileges

Category: Others

Content

In the LockingVault contract, the `getUserTier()` verification function exclusively evaluates the rigid mathematical `amount` of capital associated with a user's vault deposit, fatally omitting active temporal validations asserting whether the lock duration has expired. Because `unlockTime` is completely unenforced during Tier resolution, users are incentivized to initiate a minor 30-day foundational lock, patiently wait for literal expiration, and subsequently never invoke `unlock()`. By intentionally leaving the volatile capital inside the vault post-expiration, the user functionally transitions their locked funds into a standard fluid withdrawal account (capable of executing 0-second instant unrestricted withdrawals preventing market crashes) whilst perpetually maintaining the highest operational Diamond/Platinum Tier account privileges indefinitely across the platform. This logic systematically eradicates the foundational veTokenomics premise explicitly dictating "rigid liquidity deprivation in exchange for systemic privileges".

Code location:

LockingVault.sol#L141-149

```
    function getUserTier(address user) external view returns (uint8) {
        uint256 amount = _positions[user].amount;
        ...
     }
```

Solution

It is recommended to invalidate any administrative privilege queries linked to explicitly expired portfolios by fundamentally overriding tier evaluations natively: returning `0` utilizing `if (block.timestamp >= _positions[user].unlockTime) return 0;` at the absolute entry of the `getUserTier` function.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002604030002	SlowMist Security Team	2026.03.30 - 2026.04.03	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and the SlowMist team's analysis tool to audit the project. During the audit work, we found 1 critical risk, 3 high risks, 7 medium risks, 4 low risks, 5 suggestions, and 8 information. All other findings were fixed or acknowledged. Through on-chain contract verification, the project team has configured the corresponding role addresses as the corresponding contract addresses or deployer addresses. Both the deployer and the corresponding EOA role addresses are managed by the 2-of-3 Safeheron MPC wallet.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist