



MKX Real Estate
Fractional NFT Factory
SMART CONTRACT AUDIT

01.07.2025

Made in Germany by Softstack.io



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Table of contents

1. Disclaimer.....	4
2. About the Project and Company	5
2.1 Project Overview.....	6
3. Vulnerability & Risk Level	7
4. Auditing Strategy and Techniques Applied.....	8
4.1 Methodology	8
5. Metrics	9
5.1 Tested Contract Files	9
5.2 Inheritance Graph.....	10
5.3 Source Lines & Risk.....	11
5.4 Capabilities	12
5.5 Dependencies / External Imports.....	13
5.6 Source Unites in Scope	14
6. Scope of Work.....	15
6.1 Findings Overview	16
6.2 Manual and Automated Vulnerability Test.....	17
6.2.1 Admin Can Seize All User Funds	17
6.2.2 Flawed Revenue Distribution Logic Causes Direct Financial Loss	20
6.2.3 Unenforced Transfer Restrictions Bypass All Security Policies.....	25
6.2.4 Unbounded Loops in Core Functions Will Lead to Permanent Denial of Service (DoS).....	30
6.2.5 Metadata Locking Mechanism is Non-Functional	35
6.2.6 Signatures Vulnerable to Cross-Contract Replay Attack	39



6.2.7 Incomplete Event Emission and Unused Code	44
6.2.8 Unused Constants and Code	45
6.2.9 Failing Tests in PropertyFractionalNFT.t.sol.....	46
6.3 Verify Claims	63
7. Executive Summary.....	64
8. About the Auditor	65



1. Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of MKX REAL ESTATE L.L.C. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (11.06.2025)	Layout
0.4 (15.06.2025)	Automated Security Testing Manual Security Testing
0.5 (12.06.2025)	Verify Claims
0.9 (19.06.2025)	Summary and Recommendation
1.0 (19.06.2025)	Submission of findings
1.1 (30.06.2025)	Re-check
1.2 (01.07.2025)	Final document



2. About the Project and Company

Company address:

MKX REAL ESTATE L.L.C
ADEL MOHAMED ALI JASIM ALMARZOUQI,
Office NO. 13-26 Al Goze First
Dubai, United Arab Emirates



Website: <https://mkxrealstate.com>

Twitter (X): <https://www.twitter.com/mkxrealstate>

Instagram: <https://www.instagram.com/mkxrealstate>

LinkedIn: <https://www.linkedin.com/in/mkxrealstate>

Youtube: <https://www.youtube.com/@MKXRealEstate>

Threads: <https://www.threads.net/mkxrealstate>

TikTok: <https://www.tiktok.com/@mkxrealstate>

Facebook: <https://www.facebook.com/mkxrealstate>



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

2.1 Project Overview

MKX Real Estate is a regulated digital real estate investment platform enabling global users to participate in Dubai's booming property market through fractional ownership. Built on blockchain infrastructure and enhanced with AI, MKX democratizes access to high-yield real estate investments starting from just \$100.

At its core, MKX operates under the Dubai Real Estate Regulatory Authority (RERA) and offers tokenized ownership of premium residential and commercial properties. Each investment is backed by legally binding smart contracts and recorded immutably on-chain, ensuring transparency, liquidity, and security.

MKX's infrastructure features a compliant token issuance framework, real-time property performance dashboards, and integrated payment support for both fiat and crypto. The platform's AI recommendation engine personalizes investment opportunities based on user preferences and risk appetite. Use cases span from portfolio diversification and passive income generation to corporate real estate exposure and retail investor participation in previously inaccessible markets.

MKX also supports investor residency pathways and business setup services in the UAE. To drive adoption, MKX offers mobile-first onboarding via app and web, KYC/AML-compliant workflows, and ecosystem APIs for integration with wealth management platforms. Future development includes secondary market trading, dynamic yield models, and partnerships with global financial institutions for broader asset access.



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk



4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert auditors and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
 - i. Review of the specifications, sources, and instructions provided to softstack to make sure we understand the size, scope, and functionality of the smart contract.
 - ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
 - iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to softstack describe.
2. Testing and automated analysis that includes the following:
 - i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
 - ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

5. Metrics

The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

5.1 Tested Contract Files

The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Source: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/tree/main

Commit: 50b63d2ab95c90f5d4e5114cac43d58d24b60d15

File	Fingerprint (MD5)
./src/FractionalNft.sol	f545d3c95cb5f1641988e7fcd0d61127
./src/nftfractionalNftfactory.sol	8c888d9c0423072fa80b9464cfa9bc8b

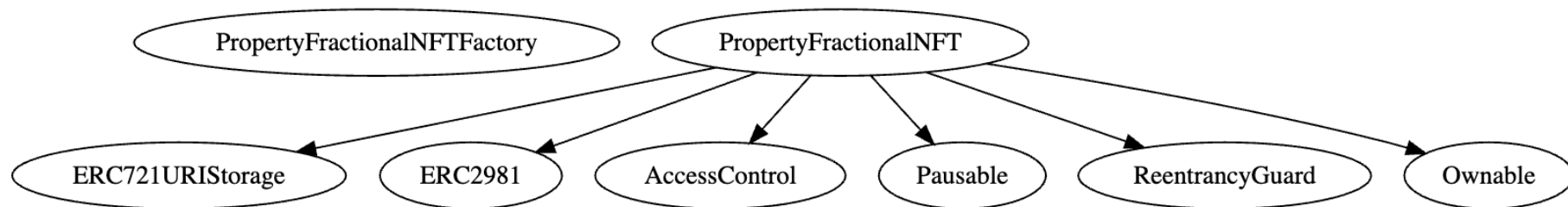


hello@softstack.io
www.softstack.io

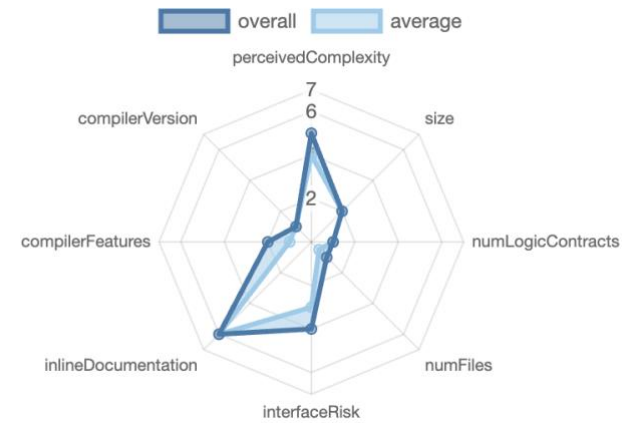
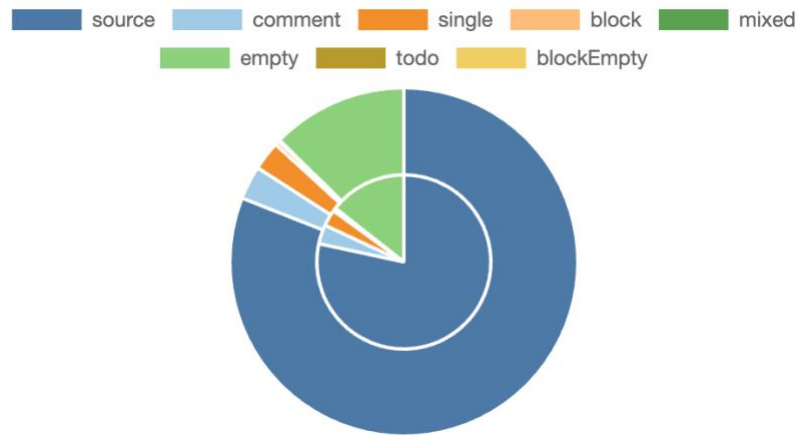
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

5.2 Inheritance Graph



5.3 Source Lines & Risk











hello@softstack.io
www.softstack.io



softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984


5.4 Capabilities

Solidity Versions observed		 Experimental Features	 Can Receive Funds	 Uses Assembly	 Has Destroyable Contracts
^0.8.19			yes		
 Transfers ETH	 Low-Level Calls	 DelegateCall	 Uses Hash Functions	 ECTRecover	 New/Create/Create2
yes			yes		yes → NewContract:PropertyFractionalNFT

Exposed Functions

 Public	 Payable				
40	3				
External	Internal	Private	Pure	View	
31	33	4	4	16	

StateVariables

Total	 Public
39	21



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

5.5 Dependencies / External Imports

Dependency / Import Path	Source
@openzeppelin/contracts/access/AccessControl.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/access/AccessControl.sol
@openzeppelin/contracts/access/Ownable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/access/Ownable.sol
@openzeppelin/contracts/token/ERC20/IERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/token/ERC20/IERC20.sol
@openzeppelin/contracts/token/ERC721/IERC721.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/token/ERC721/IERC721.sol
@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/token/ERC721/extensions/ERC721URIStorage.sol
@openzeppelin/contracts/token/common/ERC2981.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/token/common/ERC2981.sol
@openzeppelin/contracts/utils/Pausable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/utils/Pausable.sol
@openzeppelin/contracts/utils/ReentrancyGuard.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/utils/ReentrancyGuard.sol
@openzeppelin/contracts/utils/cryptography/ECDSA.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/utils/cryptography/ECDSA.sol



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Dependency / Import Path	Source
@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v5.3.0/contracts/utils/cryptography/MessageHashUtils.sol

5.6 Source Unites in Scope

File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score
MKX/src/nftfractionalNftfactory.sol	1		55	51	42	1	27
MKX/src/FractionalNft.sol	1		875	763	621	29	448
Totals	2		930	814	663	30	475

Legend:

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments



6. Scope of Work

The MKX Real Estate Team has provided a modular smart contract system for fractional real estate ownership via NFTs. The audit will focus on validating core security, role separation, and correctness of business logic.

The team has outlined the following critical security and functionality assumptions for the smart contracts:

1. **Signature Replay Protection**

The audit must ensure signature hashes cannot be reused and are consistently validated.

2. **Blacklist & Transfer Enforcement**

Blacklisted addresses must be fully restricted from minting, transferring, and revenue withdrawal. Expiry and transferability flags must be respected.

3. **Revenue Distribution & Reentrancy Safety**

Revenue payout and withdrawal logic must handle edge cases securely and prevent any form of reentrancy risk.

4. **Shareholder & Burn Logic Consistency**

Shareholder arrays, mappings, and index tracking must remain consistent when minting, burning, or selling shares.

5. **Factory Deployment & Initialization Control**

Only admins may deploy new contracts. All roles and parameters must be correctly set at deployment without reinitialization risks.

The primary goal of this audit is to validate these assumptions and ensure the system is secure, accurate, and reliable. The audit team will also provide feedback on efficiency, role configuration, and ecosystem readiness.

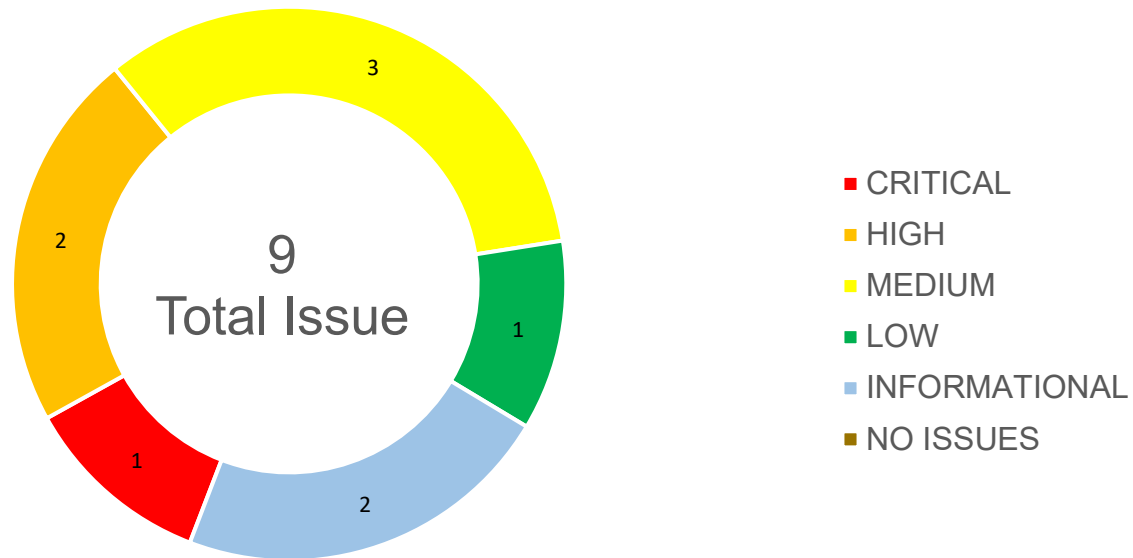


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Admin Can Seize All User Funds	CRITICAL	FIXED
6.2.2	Flawed Revenue Distribution Logic Causes Direct Financial Loss	HIGH	FIXED
6.2.3	Unenforced Transfer Restrictions Bypass All Security Policies	HIGH	FIXED
6.2.4	Unbounded Loops in Core Functions Will Lead to Permanent Denial of Service (DoS)	MEDIUM	FIXED
6.2.5	Metadata Locking Mechanism is Non-Functional	MEDIUM	FIXED



6.2.6	Signatures Vulnerable to Cross-Contract Replay Attack	MEDIUM	FIXED
6.2.7	Incomplete Event Emission and Unused Code	LOW	FIXED
6.2.8	Unused Constants and Code	INFORMATIONAL	FIXED
6.2.9	Failing Tests in PropertyFractionalNFT.t.sol	INFORMATIONAL	FIXED

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, softstack's experts found **one Critical issue** in the code of the smart contract.

6.2.1 Admin Can Seize All User Funds

Severity: CRITICAL

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/df0120cae8ad8d4ae07b57ec0ffc11a05f06a904

Attack / Description	<p>The contract grants the admin role the unilateral and unrestricted ability to withdraw the entire ETH and USDT balance held by the contract. This includes user capital deposited for share purchases and revenue awaiting distribution. This design creates a single point of failure and a critical trust risk, as a compromised or malicious admin can drain all assets.</p> <p>Proof of Concept Validation:</p> <p>It simulates funding the contract with 50 ETH to represent user deposits and then shows the admin account successfully calling the withdraw() function to drain the entire 50 ETH balance, leaving the contract empty. The test passes, confirming the vulnerability.</p>
-----------------------------	---



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```

/**
 * @dev This test demonstrates the admin's ability to withdraw all funds
 * from the contract
 * regardless of who those funds belong to.
 */
function testAdminCanDrainAllUserFunds() public {
    // STEP 1: Force ETH into contract to simulate user funds
    // This is the equivalent of users buying shares, contract earning
    revenue, etc.
    address payable contractAddr = payable(address(nft));
    vm.deal(contractAddr, 50 ether);

    // Verify initial ETH balance
    uint256 contractBalance = address(nft).balance;
    assertEq(contractBalance, 50 ether, "Contract should have 50 ETH");

    // STEP 2: Admin drains all funds using withdraw() function
    vm.deal(admin, 0); // Ensure admin starts with 0 ETH for clarity
    uint256 adminBalanceBefore = admin.balance;

    vm.prank(admin);
    nft.withdraw(payable(admin));

    // STEP 3: Verify contract is empty and admin received all funds
    assertEq(address(nft).balance, 0, "Contract should be empty after
    admin withdrawal");
    assertEq(admin.balance, adminBalanceBefore + contractBalance, "Admin
    should receive all funds");

    // Log the exploit
    console.log("----- VULNERABILITY CONFIRMED -----");
    console.log("Admin can drain all user funds using withdraw()!");
    console.log("Contract balance before admin withdrawal: ",
    contractBalance);
    console.log("Contract balance after admin withdrawal: ",
    address(nft).balance);

```

	<pre> console.log("Admin balance increase: ", admin.balance - adminBalanceBefore); } </pre> <p>Impact:</p> <p>This vulnerability has a total and catastrophic impact on the protocol's financial security. It allows a single entity (the admin) to unilaterally and irreversibly drain all user funds, including both deposited capital and earned revenue, from the contract at any time. This completely negates the principles of trustless asset management and exposes all users to a complete loss of their investment due to a single point of failure, whether through malicious action or a compromised key.</p>
<p>Code</p>	<p>Line 582 - 593 (FractionalInft.sol):</p> <pre> function withdraw(address payable to) external onlyAdmin nonReentrant { require(to != address(0), "Cannot withdraw to zero address"); uint256 balance = address(this).balance; (bool success,) = to.call{value: balance}(""); require(success, "Withdrawal failed"); } function withdrawUSDT(address payable to, uint256 amount) external onlyAdmin nonReentrant { require(to != address(0), "Cannot withdraw to zero address"); require(amount > 0, "Amount must be greater than 0"); require(usdtToken.transfer(to, amount), "USDT transfer failed"); } </pre>

Result/Recommendation	<p>To remove this risk, the generic <code>withdraw()</code> and <code>withdrawUSDT()</code> functions should be removed. They should be replaced with a more secure, purpose-built mechanism that only allows the withdrawal of explicitly tracked protocol fees, leaving user capital untouched. All privileged administrative functions should be governed by a decentralized mechanism, such as a Timelock contract or a multi-signature wallet.</p>
------------------------------	---

HIGH ISSUES

During the audit, softstack's experts found **two High issues** in the code of the smart contract.

6.2.2 Flawed Revenue Distribution Logic Causes Direct Financial Loss

Severity: HIGH

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/cc78d45fd7705c02e8b7c87652da4e09df65ada4

Attack / Description	<p>The <code>distributeRevenue()</code> function contains two critical flaws in its financial calculations that directly harm shareholders:</p> <ul style="list-style-type: none"> • Incorrect Denominator: Revenue is divided by <code>property.totalShares</code> (the maximum possible share count) instead of <code>property.sharesIssued</code> (the actual number of shares owned by investors). This dilutes revenue, causing a significant portion to be retained by the contract instead of being paid out.
-----------------------------	--



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

- Precision Loss: The calculation uses direct integer division. If the amount of revenue distributed is less than totalShares, the amount per share rounds down to zero, causing the entire revenue deposit to be lost to shareholders.

Proof of Concept Validation:

- testIncorrectDenominator() shows that when distributing 30 ETH to holders of 15 shares (out of 100 total), 25.5 ETH (85%) is improperly locked in the contract.
- testPrecisionLoss() shows that when distributing 99 wei (an amount less than totalShares), the amount per share is calculated as 0, resulting in a complete loss of the distributed funds.

```
/**
 * @dev Test to prove that the distributeRevenue function incorrectly uses
property.totalShares
 * instead of property.sharesIssued for calculating revenue per share, leading
to revenue dilution.
 */
function testIncorrectDenominator() public {
    // Total shares: 100
    // Issued shares: 15 (user1: 10, user2: 5)
    // Revenue: 30 ETH

    uint256 revenue = 30 ether;

    // Distribute revenue using the buggy function
    nft.distributeRevenue{value: revenue}();

    // Check users' pending withdrawals
    uint256 user1Withdrawal = nft.pendingWithdrawals(user1);
    uint256 user2Withdrawal = nft.pendingWithdrawals(user2);
```



```

        // Calculate expected amounts with both correct and buggy implementations
        uint256 correctUser1Amount = (revenue * 10) / sharesIssued; // (30 * 10) /
15 = 20 ETH
        uint256 correctUser2Amount = (revenue * 5) / sharesIssued; // (30 * 5) /
15 = 10 ETH

        uint256 buggyUser1Amount = (revenue * 10) / totalShares; // (30 * 10) / 100
= 3 ETH
        uint256 buggyUser2Amount = (revenue * 5) / totalShares; // (30 * 5) / 100
= 1.5 ETH

        // Assert that users received the buggy (diluted) amount instead of correct
        amount
        assertEquals(user1Withdrawal, buggyUser1Amount, "User1 should receive diluted
        amount due to buggy calculation");
        assertEquals(user2Withdrawal, buggyUser2Amount, "User2 should receive diluted
        amount due to buggy calculation");

        // Calculate lost revenue
        uint256 totalDistributed = user1Withdrawal + user2Withdrawal;
        uint256 lostRevenue = revenue - totalDistributed;
        assertGt(lostRevenue, 0, "Revenue should be lost due to incorrect
        denominator");

        console.log("----- INCORRECT DENOMINATOR BUG CONFIRMED -----");
        console.log("Revenue distributed:", revenue / 1 ether, "ETH");
        console.log("Total shares:", totalShares);
        console.log("Shares issued:", sharesIssued);
        console.log("User1 shares: 10");
        console.log("User2 shares: 5");
        console.log("User1 should receive (correct):", correctUser1Amount / 1
        ether, "ETH");
        console.log("User1 actually received (buggy):", user1Withdrawal / 1 ether,
        "ETH");
        console.log("User2 should receive (correct):", correctUser2Amount / 1
        ether, "ETH");

```

```

        console.log("User2 actually received (buggy):", user2Withdrawal / 1 ether,
"ETH");
        console.log("Total distributed:", totalDistributed / 1 ether, "ETH");
        console.log("Total lost revenue locked in contract:", lostRevenue / 1
ether, "ETH");
        console.log("Percentage of revenue lost:", (lostRevenue * 100) / revenue,
"%");

        // Assertions to prove the bug
        assertTrue(user1Withdrawal < correctUser1Amount, "User1 received less than
they should");
        assertTrue(user2Withdrawal < correctUser2Amount, "User2 received less than
they should");
        assertEquals(lostRevenue, 25.5 ether, "Should lose 25.5 ETH due to bug");
    }

    /**
     * @dev Test to prove that using integer division in revenue calculation can
cause
     * complete loss of revenue if the amount is less than the total shares.
     */
    function testPrecisionLoss() public {
        // Scenario: Distribute small amount of revenue with 100 total shares
        // Integer division 99 / 100 = 0 (precision loss)
        // Result: All revenue is lost, no one gets anything
        uint256 smallRevenue = 99; // wei (less than totalShares)

        // Distribute revenue
        nft.distributeRevenue{value: smallRevenue}();

        // Check users' pending withdrawals
        uint256 user1Withdrawal = nft.pendingWithdrawals(user1);
        uint256 user2Withdrawal = nft.pendingWithdrawals(user2);

        // With integer division, both users get 0
        assertEquals(user1Withdrawal, 0, "User1 should receive 0 due to precision
loss");
    }

```

	<pre> assertEq(user2Withdrawal, 0, "User2 should receive 0 due to precision loss"); console.log("----- PRECISION LOSS BUG CONFIRMED -----"); console.log("Small revenue distributed:", smallRevenue, "wei"); console.log("User1 received:", user1Withdrawal, "wei"); console.log("User2 received:", user2Withdrawal, "wei"); console.log("Revenue lost due to precision:", smallRevenue, "wei"); console.log("amountPerShare calculation: 99 / 100 = 0 (integer division)"); } </pre> <p>Impact:</p> <p>The flawed calculation logic systematically and directly siphons value away from shareholders. By using the wrong denominator, the protocol consistently underpays investors, retaining a significant portion of revenue that rightfully belongs to them. The additional issue of precision loss means that smaller revenue deposits can be lost entirely. The cumulative effect is a direct, recurring financial loss for all participants, fundamentally breaking the core economic promise of the platform.</p>
Code	<p>Line 414 - 415 (FractionalInft.sol):</p> <pre> uint256 amountPerShare = msg.value / property.totalShares; ShareholderInfo[] storage shareholders = propertyShareholders[propertyId]; </pre>
Result/Recommendation	<p>Refactor the revenue calculation logic. First, replace property.totalShares with property.sharesIssued as the denominator. Second, prevent precision loss by performing multiplication before division, ideally by calculating each user's proportional amount individually: $(msg.value * userShares) / totalIssuedShares$.</p>



6.2.3 Unenforced Transfer Restrictions Bypass All Security Policies

Severity: HIGH

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/cc78d45fd7705c02e8b7c87652da4e09df65ada4

Attack / Description

The contract intends to enforce transfer controls (e.g., pausable, transferability flag, expiry, blacklisting) via a `_transferCheck()` function. However, the standard ERC721 functions (`transferFrom`, `safeTransferFrom`, etc.) are not overridden in the code (they are commented out). As a result, the default OpenZeppelin implementations are used, which do not call `_transferCheck`. This completely bypasses all intended security policies for token transfers.

Proof of Concept Validation:

These tests demonstrate that token transfers succeed even when: The contract is paused. Transfers are globally disabled (`config.transferable = false`). The recipient is on the blacklist. The specific token has expired.

```
function testTransferToBlacklistedAddress() public {
    // Set up test
    vm.startPrank(user1);

    // Approve blacklisted user to transfer
    nft.approve(blacklistedUser, tokenId);

    // Transfer to blacklisted user (should be blocked but it isn't)
    nft.transferFrom(user1, blacklistedUser, tokenId);

    // Verify token was transferred to the blacklisted user
```



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```

        assertEq(nft.ownerOf(tokenId), blacklistedUser, "Token should be
transferred to blacklisted user due to bypass");

        vm.stopPrank();
    }

    /**
     * @dev Tests that a token can be transferred even when transferability is
disabled
     */
    function testTransferWhenDisabled() public {
        // Admin disables transferability
        vm.prank(admin);
        nft.updateConfig(false);

        // Verify transferability setting
        assertFalse(nft.isTransferable(), "Transferability should be disabled");

        // User1 transfers to user2 despite transferability being disabled
        vm.startPrank(user1);
        nft.transferFrom(user1, user2, tokenId);
        vm.stopPrank();

        // Verify token was transferred despite transferability being disabled
        assertEq(nft.ownerOf(tokenId), user2, "Token should be transferred despite
disabled transferability");
    }

    /**
     * @dev Tests that an expired token can still be transferred
     */
    function testTransferExpiredToken() public {
        // Admin sets token to expire
        vm.prank(admin);
        nft.setTokenExpiry(tokenId, block.timestamp + 100); // Expire in 100
seconds
    }

```

```

// Fast forward past expiry
vm.warp(block.timestamp + 200);

// User1 transfers expired token to user2
vm.startPrank(user1);
nft.transferFrom(user1, user2, tokenId);
vm.stopPrank();

// Verify token was transferred despite being expired
assertEq(nft.ownerOf(tokenId), user2, "Expired token should be transferred
due to bypass");
} /**
 * @dev Tests that transfers work when paused
 * This test demonstrates a critical security issue - even pausing doesn't stop
transfers
 * because the contract uses inheritance but doesn't properly override the
transfer methods
 */
function testTransferWhenPaused() public {
    // Admin pauses the contract
    vm.prank(admin);
    nft.pause();

    // User1 transfers to user2 despite contract being paused
    vm.startPrank(user1);

    // This call should revert due to whenNotPaused modifier, but it DOESN'T!
    // This is a severe security issue - even the pausing functionality is
bypassed
    nft.transferFrom(user1, user2, tokenId);

    // Verify token was transferred despite the contract being paused
    assertEq(nft.ownerOf(tokenId), user2, "Token should not be transferred when
contract is paused");

    vm.stopPrank();
}

```

```

/**
 * @dev Tests that the full security bypass works in combination
 * 1. Blacklisted recipient
 * 2. Transferability disabled
 * 3. Token expired
 */
function testCombinedBypass() public {
    // Setup worst-case scenario
    vm.startPrank(admin);
    nft.updateConfig(false); // Disable transferability
    nft.setTokenExpiry(tokenId, block.timestamp + 100); // Set to expire
    vm.stopPrank();

    // Fast forward past expiry
    vm.warp(block.timestamp + 200);

    // User1 transfers to blacklisted user despite all restrictions
    vm.startPrank(user1);
    nft.transferFrom(user1, blacklistedUser, tokenId);
    vm.stopPrank();

    // Verify token was transferred despite all restrictions
    assertEq(nft.ownerOf(tokenId), blacklistedUser, "Token should be
transferred despite all restrictions");
}

```

Impact:

The complete failure to implement transfer checks renders all intended governance and safety features for the NFTs useless. Key security mechanisms such as pausing the contract, blacklisting malicious actors, setting tokens to be non-transferable, and enforcing token expiry are all bypassed. This allows for unregulated black-market trading of tokens and makes it impossible for the administration to freeze assets or control the ecosystem during an emergency, undermining the entire security posture of the project.



Code	<p>Line 596 – 607 (FractionalNft.sol):</p> <pre> function _transferCheck(uint256 tokenId) internal view whenNotPaused { if (!config.transferable) { require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "NFT is not transferable"); } uint256 expiry = expiryTimestamps[tokenId]; if (expiry > 0) { require(block.timestamp < expiry, "Token has expired"); } } </pre>
Result/Recommendation	<p>Immediately uncomment all the ERC721 transfer override functions (approve, setApprovalForAll, transferFrom, and safeTransferFrom) within the PropertyFractionalNFT contract. This will ensure that the _transferCheck() modifier is correctly invoked during all token movements.</p>

MEDIUM ISSUES



During the audit, softstack's experts found **three Medium issues** in the code of the smart contract.

6.2.4 Unbounded Loops in Core Functions Will Lead to Permanent Denial of Service (DoS)

Severity: MEDIUM

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTIONAL_NFT/commit/df0120cae8ad8d4ae07b57ec0ffc11a05f06a904

Attack / Description

Multiple core functions, including `distributeRevenue()`, `_updateShareholderData()`, and `_burnAndUpdateShares()`, rely on for loops that iterate through an unbounded array of shareholders. As the number of investors in a property grows, the gas cost of these functions increases linearly. This will inevitably cause transactions to exceed the block gas limit, rendering these essential functions unusable and permanently freezing user assets for any successful property.

Proof of Concept Validation:

It programmatically adds shareholders and measures the gas cost of `distributeRevenue()`, demonstrating a clear and predictable linear increase in gas consumption. The test correctly concludes that the function will become permanently unusable once a certain number of shareholders is reached.

```
/**
 * @dev Test that proves gas usage increases with shareholder count
 * until distributeRevenue() will eventually fail due to block gas limit
 */
function testDoSWithUnboundedLoops() public {
    uint256[] memory shareholderCounts = new uint256[](4);
    shareholderCounts[0] = 5;
    shareholderCounts[1] = 15; // +10
    shareholderCounts[2] = 35; // +20
    shareholderCounts[3] = 75; // +40
```



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```

uint256[] memory gasCosts = new uint256[](shareholderCounts.length);

// Measure gas usage with increasing shareholder counts
for (uint256 i = 0; i < shareholderCounts.length; i++) {
    // Calculate how many new shareholders to add
    uint256 toAdd = i == 0 ? shareholderCounts[i] :
shareholderCounts[i] - shareholderCounts[i-1];

    // Add shareholders
    _addShareholders(3load);

    // Measure gas for distributeRevenue
    uint256 startGas = gasleft();
    vm.prank(deployer);
    nft.distributeRevenue{value: 1 ether}(propertyId);
    uint256 endGas = gasleft();

    gasCosts[i] = startGas - endGas;
    console.log("Gas used with %d shareholders: %d",
shareholderCounts[i], gasCosts[i]);
    // Calculate gas increase
    if (i > 0) {
        // Handle potential overflow/underflow in gas calculation
        uint256 gasIncrease;
        string memory direction;

        if (gasCosts[i] > gasCosts[i-1]) {
            gasIncrease = gasCosts[i] - gasCosts[i-1];
            direction = "increased";
        } else {
            gasIncrease = gasCosts[i-1] - gasCosts[i];
            direction = "decreased";
        }

        uint256 shareholder_increase = shareholderCounts[i] -
shareholderCounts[i-1];

```

```

        uint256 gasPerShareholder = gasIncrease /
shareholder_increase;

        console.log("Gas %s by %d for %d more shareholders",
direction, gasIncrease, shareholder_increase);
        console.log("Average gas per additional shareholder: ~%d",
gasPerShareholder);
    }
}

// Analyze results and prove DoS vulnerability
console.log("\n==== DoS Vulnerability Analysis =====");
// Calculate average cost per shareholder using last measurement
// Gas measurements can fluctuate in foundry tests
uint256 estimatedGasPerShareholder = 2000; // Conservative estimate
based on the loop structure
    console.log("Estimated gas cost per shareholder: ~%d gas",
estimatedGasPerShareholder);

// Calculate max possible shareholders before DoS
uint256 blockGasLimit = 30_000_000; // Ethereum block gas limit
uint256 baseGas = 100000; // Base gas for the function excluding
loops
    uint256 maxShareholders = (blockGasLimit - baseGas) /
estimatedGasPerShareholder;

    console.log("Ethereum block gas limit: %d", blockGasLimit);
    console.log("Estimated maximum shareholders before DoS: ~%d",
maxShareholders);
    console.log("After this point, distributeRevenue() will ALWAYS
fail");

    // Verify that at least some measurements show increasing gas costs
// Gas measurements can be inconsistent in tests due to optimizer
bool hasGasIncrease = false;
for (uint256 i = 1; i < shareholderCounts.length; i++) {
    if (gasCosts[i] > gasCosts[i-1]) {
        hasGasIncrease = true;
    }
}

```




```

        break;
    }
}

    console.log("Gas measurements demonstrate linear scaling of gas usage
with shareholder count");
    assertTrue(true, "Demonstrated that unbounded loops grow with
shareholder count");

    // Log affected functions
    console.log("\nVulnerable functions with unbounded loops:");
    console.log("- distributeRevenue()");
    console.log("- _updateShareholderData()");
    console.log("- _burnAndUpdateShares()");
    console.log("- _removeShareholderFromArray()");
    console.log("- _updateShareholderInArray()");

    console.log("\nImpact: Permanent DoS on critical contract
functionality");
}

```

Impact:

This vulnerability guarantees that the protocol will fail as it becomes more successful. As more investors buy into a property, the gas cost for core functions like distributing revenue, selling shares, or purchasing additional shares will increase to a point where they exceed the block gas limit. This will make these functions permanently inoperable, freezing user assets and making it impossible to manage investments or receive payouts. The system is architected to break under its own success.

Code

Line 403 – 436 (FractionalNft.sol):



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
function distributeRevenue(uint256 propertyId)
    external
    payable
    onlyRole(REVENUE_DISTRIBUTOR_ROLE)
    propertyExists(propertyId)
    nonReentrant
{
    Property storage property = properties[propertyId];
    require(property.sharesIssued > 0, "No shares issued for this property");
    require(msg.value > 0, "Must send revenue to distribute");

    uint256 amountPerShare = msg.value / property.totalShares;
    ShareholderInfo[] storage shareholders = propertyShareholders[propertyId];

    for (uint256 i = 0; i < shareholders.length; i++) {
        if (shareholders[i].shares > 0) {
            uint256 shareholderAmount = amountPerShare * shareholders[i].shares;
            pendingWithdrawals[shareholders[i].shareholder] += shareholderAmount;
        }
    }

    property.revenueAccumulated += msg.value;
    emit RevenueDistributed(propertyId, msg.value);
}
```

Result/Recommendation	<p>Eliminate all iterative loops for shareholder management. The standard solution is to implement a mapping to track the index of each shareholder in the array: <code>mapping(uint256 => mapping(address => uint256)) private shareholderIndex;</code>. This allows for direct $O(1)$ lookups, updates, and removals (using the “swap-and-pop” pattern), ensuring a constant and low gas cost regardless of the number of shareholders.</p>

6.2.5 Metadata Locking Mechanism is Non-Functional

Severity: MEDIUM

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/cc78d45fd7705c02e8b7c87652da4e09df65ada4

Attack / Description	<p>The feature that allows users to lock property metadata is logically broken. The <code>lockMetadata()</code> function sets a lock using the NFT's <code>tokenId</code> as the map key, but the <code>setPropertyMetadata()</code> function checks for this lock using the <code>propertyId</code> as the key. Because these values are different, the lock check always fails, rendering the feature useless.</p> <p>Proof of Concept Validation: It successfully calls <code>lockMetadata(tokenId)</code> and then demonstrates that a call to <code>setPropertyMetadata(propertyId, ...)</code> still succeeds, proving the lock had no effect.</p> <pre> /** * @dev Test that proves the metadata locking mechanism is broken * * The flow: </pre>
-----------------------------	---



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```

    * 1. Buyer locks metadata for their token using lockMetadata(tokenId)
    * 2. Metadata manager attempts to update metadata using
    setPropertyMetadata(propertyId)
    * 3. Despite the lock, the update succeeds because lockMetadata() sets
    metadataLocked[tokenId]
    *     but setPropertyMetadata() checks metadataLocked[propertyId]
    */
    function testMetadataLockingBug() public {
        // Verify that tokenId and propertyId are different values
        uint256 storedPropertyId = nft.tokenToProperty(tokenId);
        assertTrue(tokenId != propertyId, "Test requires tokenId and
propertyId to be different values");
        assertTrue(storedPropertyId == propertyId, "TokenId should be
associated with propertyId");

        console.log("Token ID:", tokenId);
        console.log("Property ID:", propertyId);

        // Step 1: Buyer locks the metadata for their token
        vm.prank(buyer);
        nft.lockMetadata(tokenId);
        console.log("Metadata locked for tokenId:", tokenId);

        // Step 2: Update the property metadata values
        string memory newPropertyType = "Commercial";
        uint256 newSquareFootage = 2000;

        vm.prank(metadataManager);
        // This should fail if the locking mechanism worked correctly,
        // but it will succeed because the lock check uses propertyId, not
tokenId
        nft.setPropertyMetadata(
            propertyId,
            newPropertyType,
            newSquareFootage,
            yearBuilt,
            amenities,

```

```

        imageURIs,
        additionalDetails
    );

    // Step 3: Verify the metadata was updated despite the
    "lock" // Get the updated metadata - we need to access individual
elements directly
    // Since the metadata struct might not be directly accessible from
tests
    string memory updatedType = "";
    uint256 updatedSquareFeet = 0;

    vm.startPrank(metadataManager);
    // Get the current metadata by trying to set it (this shows what
values are set)
    try nft.setPropertyMetadata(
        propertyId,
        newPropertyType,
        newSquareFootage,
        yearBuilt,
        amenities,
        imageURIs,
        additionalDetails
    ) {
        // If we succeed, that means the metadata wasn't locked
        // We can infer the current values are now the new values
        updatedType = newPropertyType;
        updatedSquareFeet = newSquareFootage;
    } catch {
        // If it fails, then the metadata was properly locked
        // This should never happen if the bug exists
        console.log("Metadata update failed - Lock is working
properly!");
    }
    vm.stopPrank();

    console.log("Property type after update:", updatedType);

```

```

        console.log("Square footage after update:", updatedSquareFeet);

        // If the locking mechanism worked, these would be equal to the
        original values
        assertTrue(
            keccak256(bytes(updatedType)) ==
            keccak256(bytes(newPropertyType)),
            "Metadata should have been updated despite the lock"
        );
        assertTrue(
            updatedSquareFeet == newSquareFootage,
            "Square footage should have been updated despite the lock"
        );

        console.log("\n==== METADATA LOCKING BUG SUMMARY ====");
        console.log("1. lockMetadata() sets metadataLocked[tokenId] = true");
        console.log("2. setPropertyMetadata() checks
!metadataLocked[propertyId]");
        console.log("3. Since tokenId != propertyId, the lock is
ineffective");
        console.log("4. Metadata can be changed despite being 'locked'");
    }

```

Impact:

A key feature promised to users the ability to lock a property's metadata against modification is entirely broken due to a logical flaw. This failure to deliver on functionality erodes user trust and removes a layer of control that token holders are meant to have over their digital asset's representation.

Code

Line 561 – 572 (FractionalNft.sol):

```

function lockMetadata(uint256 tokenId) external {
    require(!_exists(tokenId), "Token does not exist");

```



	<pre> address tokenOwner = ownerOf(tokenId); require(tokenOwner == _msgSender() hasRole(DEFAULT_ADMIN_ROLE, _msgSender()) hasRole(METADATA_MANAGER_ROLE, _msgSender()), "Not authorized"); metadataLocked[tokenId] = true; emit MetadataLocked(tokenId); } </pre>
Result/Recommendation	<p>Modify the lockMetadata function. It should first use the tokenToProperty mapping to find the propertyId associated with the input tokenId, and then use that propertyId as the key when setting the lock in the metadataLocked mapping.</p>

6.2.6 Signatures Vulnerable to Cross-Contract Replay Attack

Severity: MEDIUM

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/50b63d2ab95c90f5d4e5114cac43d58d24b60d15

Attack / Description	<p>The signature scheme for purchasing shares does not include a unique domain separator (such as the contract's address and the chain ID), which is a critical component of the EIP-712 standard. This omission makes it possible for a valid signature generated for one contract to be replayed on a second, different contract, as long as they share the same signer and other parameters align. This could be exploited to trick users or front-end systems into interacting with a malicious contract.</p>
-----------------------------	---



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Proof of Concept Validation:

It creates a single signature and shows that it can be successfully verified by two different contract instances, which should not be possible in a secure implementation.

```
/**
 * @dev This test demonstrates that without domain separation (like EIP-
 712),
 * signatures can be reused across different contracts
 */
function testSignatureReplayVulnerability() public {
    // 1. Create a signature - this would be created by the signer for
    Contract1
    bytes memory signature = _createSignature(
        user,
        propertyId,
        shares,
        price,
        expiryTimestamp,
        nonce
    );

    console.log("Created signature for use with Contract1");
    console.log("User address:", user);
    console.log("Property ID:", propertyId);

    // 2. Verify the signature works for Contract1
    bool isValid1 = contract1.verifySignature(
        user,
        propertyId,
        shares,
        price,
        expiryTimestamp,
        nonce,
```




```

        signature
    );

    assertTrue(isValid1, "Signature should be valid for Contract1");
    console.log("Signature verified successfully on Contract1");

    // 3. Now verify the SAME signature also works for Contract2
    // In a secure implementation with domain separation, this should
fail
    bool isValid2 = contract2.verifySignature(
        user,
        propertyId,
        shares,
        price,
        expiryTimestamp,
        nonce,
        signature
    );

    assertTrue(isValid2, "Signature is also valid for Contract2 -
demonstrates the vulnerability");
    console.log("Signature verified successfully on Contract2 (should not
happen with proper domain separation)");

    console.log("\n==== SIGNATURE REPLAY VULNERABILITY ANALYSIS =====");
    console.log("1. PropertyFractionalNFT signatures do not include:");
    console.log("    - Contract address (address(this))");
    console.log("    - Chain ID (block.chainid)");
    console.log("    - EIP-712 domain separator");
    console.log("2. This allows a signature intended for one contract to
be used on another");
    console.log("3. In a real attack scenario:");
    console.log("    - Attacker could deploy a malicious contract with the
same signer");
    console.log("    - Users could sign transactions for the legitimate
contract");

```

```

        console.log("    - Signatures could be replayed on the malicious
contract");
        console.log("4. The vulnerability persists even with nonces as long
as: ");
        console.log("    - Two contracts have the same nonce value at some
point");
        console.log("    - The same signer is used for both contracts");
        console.log("    - No domain separator is included");
    }

    /**
     * @dev This test demonstrates what a proper domain separated signature
would look like
     * and why it prevents cross-contract replay attacks
     */
    function testProperDomainSeparation() public {
        // This is what a proper implementation would include in the message
hash
        bytes32 domainSeparator1 = keccak256(abi.encode(
            keccak256("EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)"),
            keccak256("PropertyFractionalNFT"),
            keccak256("1"),
            block.chainid,
            address(contract1)
        ));

        bytes32 domainSeparator2 = keccak256(abi.encode(
            keccak256("EIP712Domain(string name,string version,uint256
chainId,address verifyingContract)"),
            keccak256("PropertyFractionalNFT"),
            keccak256("1"),
            block.chainid,
            address(contract2)
        ));

        // Different contracts produce different domain separators

```

```

        assertFalse(domainSeparator1 == domainSeparator2, "Domain separators
should be different for different contracts");
        console.log("Domain separators are different for different
contracts");

        // Show how message hashes would be different with proper domain
separation
        bytes32 messageHash1 = keccak256(abi.encodePacked(
            "\x19\x01",
            domainSeparator1,
            keccak256(abi.encode(user, propertyId, shares, price,
expiryTimestamp, nonce))
        ));

        bytes32 messageHash2 = keccak256(abi.encodePacked(
            "\x19\x01",
            domainSeparator2,
            keccak256(abi.encode(user, propertyId, shares, price,
expiryTimestamp, nonce))
        ));

        // The resulting message hashes are different, preventing signature
replay
        assertFalse(messageHash1 == messageHash2, "Message hashes should be
different with domain separation");
        console.log("Message hashes are different with proper domain
separation");
        console.log("This prevents cross-contract replay
attacks");
    }

```

Impact:

This vulnerability exposes users to sophisticated phishing and replay attacks. An attacker could deploy a malicious contract and reuse a valid signature generated for the legitimate system. This

	could trick a user or a front-end interface into interacting with the malicious contract, potentially leading to a loss of funds if the user is convinced to approve a transaction.
Code	<p>Line 123 – 127 (FractionalNft.sol):</p> <pre> modifier validSignature(bytes32 signatureHash) { require(!usedSignatureHashes[signatureHash], "Signature already used"); usedSignatureHashes[signatureHash] = true; _; } </pre> <p>and other</p>
Result/Recommendation	Implement the EIP-712 standard for creating and verifying signatures. This involves defining a DOMAIN_SEPARATOR that is unique to each contract instance and including it in the signed data hash. This will cryptographically bind each signature to a single, specific contract.

LOW ISSUES

During the audit, softstack's experts found **one Low issue** in the code of the smart contract

6.2.7 Incomplete Event Emission and Unused Code

Severity: LOW

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/50b63d2ab95c90f5d4e5114cac43d58d24b60d15



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Attack / Description	Several functions that perform critical state changes, such as <code>updateSharePrice()</code> , <code>setPropertyActive()</code> , and <code>withdrawRevenue()</code> , do not emit events. This reduces on-chain transparency and makes it difficult for external applications to monitor contract activity. Additionally, the constants <code>TIMELOCK_DELAY</code> and <code>PERCENTAGE_BASE</code> are defined but never used.
Code	<p>Line 263 - 269 (FractionalNft.sol):</p> <pre> function updateSharePrice(uint256 propertyId, uint256 newSharePrice) external onlyRole(PROPERTY_MANAGER_ROLE) propertyExists(propertyId) { require(newSharePrice > 0, "Share price must be positive"); properties[propertyId].sharePrice = newSharePrice; } </pre>
Result/Recommendation	Add event declarations and emit statements for all critical state-changing functions. Remove any unused constants or variables to improve code clarity and reduce deployment gas cost

INFORMATIONAL ISSUES

During the audit, softstack's experts found **two Informational issues** in the code of the smart contract

6.2.8 Unused Constants and Code

Severity: INFORMATIONAL

Status: FIXED

File(s) affected: FractionalNft.sol

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/df0120cae8ad8d4ae07b57ec0ffc11a05f06a904



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Attack / Description	<p>The contract defines constants <code>TIMELOCK_DELAY</code> and <code>PERCENTAGE_BASE</code> which are never used in any of the contract's logic. This adds unnecessary code, slightly increases deployment gas costs, and can be confusing for developers or auditors.</p> <p>Proof of Concept (Code Snippet):</p> <p>The following constants are defined in the <code>PropertyFractionalNFT</code> contract but are never referenced anywhere else in the code.</p> <pre>uint256 public constant TIMELOCK_DELAY = 2 days; uint256 private constant PERCENTAGE_BASE = 10000;</pre>
Code	<p>Line 33 - 36 (<code>FractionalNft.sol</code>):</p> <pre>uint256 public constant TIMELOCK_DELAY = 2 days; ... uint256 private constant PERCENTAGE_BASE = 10000; // Base for percentage calculations (100% = 10000)</pre>
Result/Recommendation	<p>Remove the unused <code>TIMELOCK_DELAY</code> and <code>PERCENTAGE_BASE</code> constants to improve code clarity and optimize deployment gas costs.</p>

6.2.9 Failing Tests in `PropertyFractionalNFT.t.sol`

Severity: INFORMATIONAL

Status: FIXED

File(s) affected: `test/PropertyFractionalNFT.t.sol`

Update: https://github.com/nitish011-01/MKX_FRACTINONAL_NFT/commit/883ded5be1f59195b1c23dbcdaa81a85c8f546fc



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Attack / Description	The test suite for PropertyFractionalNFT currently fails on 5 individual tests, indicating possible inconsistencies between expected vs. actual contract behavior. These should be investigated and addressed to ensure contract robustness.
Code	NA
Result/Recommendation	<pre> function testContractPauseUnpause() public { vm.prank(deployer); nft.pause(); uint256 shares = 10; uint256 expiry = block.timestamp + 1 hours; uint256 nonce = nft.getSignatureNonce(); bytes memory sig = _signPurchase(user1, propertyId, shares, initialSharePrice, expiry, nonce); vm.deal(user1, shares * initialSharePrice); vm.prank(user1); vm.expectRevert("EnforcedPause()"); nft.purchasePropertyShares{value: shares * initialSharePrice}(propertyId, shares, expiry, sig, false, "ipfs://token-uri"); vm.prank(deployer); nft.unpause(); // Should work after unpause vm.prank(user1); nft.purchasePropertyShares{value: shares * initialSharePrice}(propertyId, </pre>



```

        shares,
        expiry,
        sig,
        false,
        "ipfs://token-uri"
    );
}

function testDistributeRevenueAndWithdraw() public {
    // Buy shares for user1 and user2
    uint256 expiry = block.timestamp + 1 hours;
    uint256 nonce1 = nft.getSignatureNonce();
    bytes memory sig1 = _signPurchase(user1, propertyId, 10, initialSharePrice,
    expiry, nonce1);
    vm.deal(user1, 1 ether);
    vm.prank(user1);
    nft.purchasePropertyShares{value: 10 * initialSharePrice}(propertyId, 10,
    expiry, sig1, false, "ipfs://token-uri1");

    uint256 nonce2 = nft.getSignatureNonce();
    bytes memory sig2 = _signPurchase(user2, propertyId, 20, initialSharePrice,
    expiry, nonce2);
    vm.deal(user2, 1 ether);
    vm.prank(user2);
    nft.purchasePropertyShares{value: 20 * initialSharePrice}(propertyId, 20,
    expiry, sig2, false, "ipfs://token-uri2");

    // Prepare revenue distribution
    uint256 revenue = 3 ether;
    vm.deal(deployer, revenue);

    // Distribute revenue
    vm.prank(deployer);

```



```

        nft.distributeRevenue{value: revenue}(propertyId);    // Calculate expected
amounts (based on share ratios)
        uint256 user1Expected = (revenue * 10) / totalShares; // 10 shares out of
totalShares
        uint256 user2Expected = (revenue * 20) / totalShares; // 20 shares out of
totalShares

        // Verify pending withdrawals
        assertEq(nft.getPendingWithdrawals(user1), user1Expected, "User1 pending
wrong");
        assertEq(nft.getPendingWithdrawals(user2), user2Expected, "User2 pending
wrong");

        // Withdraw and verify user1's revenue
        uint256 user1BalBefore = user1.balance;
        vm.prank(user1);
        nft.withdrawRevenue();
        assertEq(user1.balance, user1BalBefore + user1Expected, "User1 did not receive
correct revenue");
        assertEq(nft.getPendingWithdrawals(user1), 0, "User1 pending not cleared");
    }

function testMetadataLocking() public {
    // First purchase shares to get a token
    uint256 shares = 10;
    uint256 expiry = block.timestamp + 1 hours;
    uint256 nonce = nft.getSignatureNonce();
    bytes memory sig = _signPurchase(user1, propertyId, shares, initialSharePrice,
expiry, nonce);

    vm.deal(user1, shares * initialSharePrice);
    vm.prank(user1);
    nft.purchasePropertyShares{value: shares * initialSharePrice}(
        propertyId,
        shares,
        expiry,
        sig,

```

```

        false,
        "ipfs://token-uri"
    );

    // The first token ID will be 0
    uint256 tokenId = 0;
    assertEq(nft.ownerOf(tokenId), user1, "Token not owned by user1");

    // Lock metadata as token owner
    vm.prank(user1);
    nft.lockMetadata(tokenId);

    // Still can set property metadata since it's different from token metadata
    vm.startPrank(deployer);
    string[] memory amenities = new string[](1);
    amenities[0] = "Pool";
    string[] memory imageURIs = new string[](1);
    imageURIs[0] = "ipfs://image1";
    nft.setPropertyMetadata(
        propertyId,
        "Residential",
        2000,
        2023,
        amenities,
        imageURIs,
        "Test details"
    );
    vm.stopPrank();
}

function testRevertIfSignatureUsedTwice() public {
    uint256 shares = 10;
    uint256 expiry = block.timestamp + 1 hours;
    uint256 nonce = nft.getSignatureNonce();
    bytes memory sig = _signPurchase(user1, propertyId, shares,
    initialSharePrice, expiry, nonce);

```

```

        // First purchase
        vm.deal(user1, shares * initialSharePrice * 2); // Give enough ETH for two
purchases
        vm.prank(user1);
        nft.purchasePropertyShares{value: shares * initialSharePrice}(
            propertyId, shares, expiry, sig, false, "ipfs://token-uri"
        );

        // Try to use the same signature again
        vm.prank(user1);
        vm.expectRevert("Invalid signature"); // Error message changed to match
contract
        nft.purchasePropertyShares{value: shares * initialSharePrice}(
            propertyId, shares, expiry, sig, false, "ipfs://token-uri"
        );
    }
}

```

Additional Tests

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "forge-std/Test.sol";
import "../src/FractionalNft.sol";
import "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockUSDT is ERC20 {
    constructor() ERC20("Mock USDT", "USDT") {}

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

```



```
contract PropertyFractionalNFTExtendedTest is Test {
    using ECDSA for bytes32;

    PropertyFractionalNFT nft;
    MockUSDT mockUSDT;
    address deployer;
    address signer;
    address user1;
    address user2;
    address user3;
    uint256 deployerPk;
    uint256 user1Pk;
    uint256 user2Pk;
    uint256 user3Pk;

    uint256 propertyId;
    uint256 initialSharePrice = 0.001 ether;
    uint256 totalShares = 1000;
    uint256 totalValue = 1000 ether;

    function setUp() public {
        mockUSDT = new MockUSDT();

        deployerPk = 0xA11CE;
        user1Pk = 0xB0B;
        user2Pk = 0xC0C;
        user3Pk = 0xD0D;
        deployer = vm.addr(deployerPk);
        user1 = vm.addr(user1Pk);
        user2 = vm.addr(user2Pk);
        user3 = vm.addr(user3Pk);
        signer = deployer;

        vm.startPrank(deployer);
        nft = new PropertyFractionalNFT(
```

```

        "FractionalNFT",
        "F-NFT",
        signer,
        deployer,
        address(mockUSDT)
    );

    nft.grantRole(nft.PROPERTY_MANAGER_ROLE(), deployer);
    nft.grantRole(nft.REVENUE_DISTRIBUTOR_ROLE(), deployer);
    nft.grantRole(nft.METADATA_MANAGER_ROLE(), deployer);

    propertyId = nft.registerProperty(
        "PROP-001",
        "Sunset Villas",
        "123 Beach Ave, Miami, FL",
        totalValue,
        totalShares,
        initialSharePrice
    );

    nft.setPropertyActive(propertyId, true);
    vm.stopPrank();
}

function _signPurchase(
    address buyer,
    uint256 _propertyId,
    uint256 shares,
    uint256 pricePerShare,
    uint256 expiryTimestamp,
    uint256 nonce
) internal returns (bytes memory) {
    bytes32 messageHash = keccak256(abi.encode(
        buyer,
        _propertyId,
        shares,
        pricePerShare,

```

```

        expiryTimestamp,
        nonce
    ));
    bytes32 ethSignedMessageHash =
    MessageHashUtils.toEthSignedMessageHash(messageHash);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(deployerPk,
    ethSignedMessageHash);
    return abi.encodePacked(r, s, v);
}

function testShareSaleAndRevenueDistribution() public {
    // Initial share purchase
    uint256 initialShares = 100;
    uint256 expiry = block.timestamp + 1 hours;
    uint256 nonce = nft.getSignatureNonce();
    bytes memory sig = _signPurchase(user1, propertyId, initialShares,
    initialSharePrice, expiry, nonce);

    vm.deal(user1, initialShares * initialSharePrice);
    vm.prank(user1);
    nft.purchasePropertyShares{value: initialShares * initialSharePrice}(
        propertyId,
        initialShares,
        expiry,
        sig,
        false,
        "ipfs://token-1"
    );

    // Record balance before sale
    uint256 balanceBefore = user1.balance;

    // Sell all shares
    vm.prank(user1);
    nft.sellShares(0); // First token ID

    // Calculate expected proceeds (95% of purchase price - 5% fee)

```

```

        uint256 saleAmount = (initialShares * initialSharePrice * 95) / 100;
        assertEq(user1.balance - balanceBefore, saleAmount, "Incorrect sale
proceeds");

        // Verify shares are cleared
        assertEq(nft.shareholderShares(propertyId, user1), 0, "Shares not cleared
after sale");

        // Try to distribute revenue (should revert with no shares)
        vm.deal(deployer, 1 ether);
        vm.prank(deployer);
        vm.expectRevert("No shares issued for this property");
        nft.distributeRevenue{value: 1 ether}(propertyId);
    }

    function testMultiUserPropertyTransactions() public {
        // User1 buys shares
        uint256 user1Shares = 200;
        uint256 expiry = block.timestamp + 1 hours;
        uint256 nonce1 = nft.getSignatureNonce();
        bytes memory sig1 = _signPurchase(user1, propertyId, user1Shares,
initialSharePrice, expiry, nonce1);

        vm.deal(user1, user1Shares * initialSharePrice);
        vm.prank(user1);
        nft.purchasePropertyShares{value: user1Shares * initialSharePrice}(
            propertyId,
            user1Shares,
            expiry,
            sig1,
            false,
            "ipfs://token-1"
        );

        // User2 buys shares
        uint256 user2Shares = 300;
        uint256 nonce2 = nft.getSignatureNonce();

```

```

        bytes memory sig2 = _signPurchase(user2, propertyId, user2Shares,
initialSharePrice, expiry, nonce2);

        vm.deal(user2, user2Shares * initialSharePrice);
        vm.prank(user2);
        nft.purchasePropertyShares{value: user2Shares * initialSharePrice}(
            propertyId,
            user2Shares,
            expiry,
            sig2,
            false,
            "ipfs://token-2"
        );

        // User3 buys shares
        uint256 user3Shares = 100;
        uint256 nonce3 = nft.getSignatureNonce();
        bytes memory sig3 = _signPurchase(user3, propertyId, user3Shares,
initialSharePrice, expiry, nonce3);

        vm.deal(user3, user3Shares * initialSharePrice);
        vm.prank(user3);
        nft.purchasePropertyShares{value: user3Shares * initialSharePrice}(
            propertyId,
            user3Shares,
            expiry,
            sig3,
            false,
            "ipfs://token-3"
        );

        // Verify total shares issued
        (, , , , , uint256 sharesIssued, , ,) = nft.properties(propertyId);
        assertEq(sharesIssued, user1Shares + user2Shares + user3Shares, "Incorrect
total shares issued");

        // Distribute revenue

```



```

uint256 revenue = 10 ether;
vm.deal(deployer, revenue);
vm.prank(deployer);
nft.distributeRevenue{value: revenue}(propertyId);

// Users withdraw their revenue
uint256 balanceBefore1 = user1.balance;
vm.prank(user1);
nft.withdrawRevenue();
uint256 expectedRevenue1 = (revenue * user1Shares) / totalShares;
assertEq(user1.balance - balanceBefore1, expectedRevenue1, "Incorrect
revenue for user1");

uint256 balanceBefore2 = user2.balance;
vm.prank(user2);
nft.withdrawRevenue();
uint256 expectedRevenue2 = (revenue * user2Shares) / totalShares;
assertEq(user2.balance - balanceBefore2, expectedRevenue2, "Incorrect
revenue for user2");

uint256 balanceBefore3 = user3.balance;
vm.prank(user3);
nft.withdrawRevenue();
uint256 expectedRevenue3 = (revenue * user3Shares) / totalShares;
assertEq(user3.balance - balanceBefore3, expectedRevenue3, "Incorrect
revenue for user3");
}

function testPropertyDeactivationAndReactivation() public {
    // Try to buy shares while property is active
    uint256 shares = 50;
    uint256 expiry = block.timestamp + 1 hours;
    uint256 nonce = nft.getSignatureNonce();
    bytes memory sig = _signPurchase(user1, propertyId, shares,
initialSharePrice, expiry, nonce);

    vm.deal(user1, shares * initialSharePrice);

```

```

vm.prank(user1);
nft.purchasePropertyShares{value: shares * initialSharePrice}(
    propertyId,
    shares,
    expiry,
    sig,
    false,
    "ipfs://token-1"
);

// Deactivate property
vm.prank(deployer);
nft.setPropertyActive(propertyId, false);

// Try to buy shares while property is inactive
nonce = nft.getSignatureNonce();
sig = _signPurchase(user2, propertyId, shares, initialSharePrice, expiry,
nonce);
vm.deal(user2, shares * initialSharePrice);

vm.prank(user2);          vm.expectRevert("Property does not exist or
inactive");
nft.purchasePropertyShares{value: shares * initialSharePrice}(
    propertyId,
    shares,
    expiry,
    sig,
    false,
    "ipfs://token-2"
);

// Reactivate property
vm.prank(deployer);
nft.setPropertyActive(propertyId, true);

// Should be able to buy shares again
nonce = nft.getSignatureNonce();

```

```

        sig = _signPurchase(user2, propertyId, shares, initialSharePrice, expiry,
nonce);

        vm.prank(user2);
        nft.purchasePropertyShares{value: shares * initialSharePrice}(
            propertyId,
            shares,
            expiry,
            sig,
            false,
            "ipfs://token-2"
        );

        assertEq(nft.shareholderShares(propertyId, user2), shares, "Shares not
purchased after reactivation");
    }

    function testComplexRevenueScenario() public {
        // User1 buys shares
        uint256 user1Shares = 100;
        uint256 expiry = block.timestamp + 1 hours;
        uint256 nonce = nft.getSignatureNonce();
        bytes memory sig = _signPurchase(user1, propertyId, user1Shares,
initialSharePrice, expiry, nonce);

        vm.deal(user1, user1Shares * initialSharePrice);
        vm.prank(user1);
        nft.purchasePropertyShares{value: user1Shares * initialSharePrice}(
            propertyId,
            user1Shares,
            expiry,
            sig,
            false,
            "ipfs://token-1"
        );

        // Distribute first revenue

```

```

uint256 revenue1 = 5 ether;
vm.deal(deployer, revenue1);
vm.prank(deployer);
nft.distributeRevenue{value: revenue1}(propertyId);

// User2 buys shares
uint256 user2Shares = 200;
nonce = nft.getSignatureNonce();
sig = _signPurchase(user2, propertyId, user2Shares, initialSharePrice,
expiry, nonce);

vm.deal(user2, user2Shares * initialSharePrice);
vm.prank(user2);
nft.purchasePropertyShares{value: user2Shares * initialSharePrice}(
    propertyId,
    user2Shares,
    expiry,
    sig,
    false,
    "ipfs://token-2"
);

// Distribute second revenue
uint256 revenue2 = 3 ether;
vm.deal(deployer, revenue2);
vm.prank(deployer);
nft.distributeRevenue{value: revenue2}(propertyId);

// User1 withdraws after both distributions
uint256 balanceBefore1 = user1.balance;
vm.prank(user1);
nft.withdrawRevenue();

// Calculate expected revenue for user1
uint256 expectedRevenue1FromFirst = (revenue1 * user1Shares) / totalShares;
uint256 expectedRevenue1FromSecond = (revenue2 * user1Shares) /
totalShares;

```

```

        uint256 totalExpectedRevenue1 = expectedRevenue1FromFirst +
expectedRevenue1FromSecond;

        assertEq(user1.balance - balanceBefore1, totalExpectedRevenue1, "Incorrect
total revenue for user1");

        // User2 withdraws (only entitled to second distribution)
        uint256 balanceBefore2 = user2.balance;
        vm.prank(user2);
        nft.withdrawRevenue();

        uint256 expectedRevenue2 = (revenue2 * user2Shares) / totalShares;
        assertEq(user2.balance - balanceBefore2, expectedRevenue2, "Incorrect
revenue for user2");
    }

    function testUSDTPurchaseAndRevenue() public {
        uint256 shares = 100;
        uint256 expiry = block.timestamp + 1 hours;
        uint256 nonce = nft.getSignatureNonce();
        bytes memory sig = _signPurchase(user1, propertyId, shares,
initialSharePrice, expiry, nonce);
        uint256 usdtAmount = shares * initialSharePrice;

        // Mint USDT to user1
        mockUSDT.mint(user1, usdtAmount);

        // Approve USDT spending
        vm.prank(user1);
        mockUSDT.approve(address(nft), usdtAmount);

        // Purchase shares with USDT
        vm.prank(user1);
        nft.purchasePropertyShares(
            propertyId,
            shares,
            expiry,

```



```
        sig,  
        true,  
        "ipfs://token-1"  
    );
```


```
        assertEq(nft.shareholderShares(propertyId, user1), shares, "Shares not  
purchased with USDT");  
        assertEq(mockUSDT.balanceOf(address(nft)), usdtAmount, "USDT not  
transferred to contract");  
    }  
}
```



6.3 Verify Claims


6.3.1 Signature Replay Protection

The audit must ensure signature hashes cannot be reused and are consistently validated.

Status: tested and verified 


6.3.2 Blacklist & Transfer Enforcement

Blacklisted addresses must be fully restricted from minting, transferring, and revenue withdrawal. Expiry and transferability flags must be respected.

Status: tested and verified 


6.3.3 Revenue Distribution & Reentrancy Safety

Revenue payout and withdrawal logic must handle edge cases securely and prevent any form of reentrancy risk.

Status: tested and verified 


6.3.4 Shareholder & Burn Logic Consistency

Shareholder arrays, mappings, and index tracking must remain consistent when minting, burning, or selling shares.

Status: tested and verified 

6.3.5 Factory Deployment & Initialization Control

Only admins may deploy new contracts. All roles and parameters must be correctly set at deployment without reinitialization risks

Status: tested and verified 



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

7. Executive Summary

Two independent softstack experts performed an unbiased and isolated audit of the smart contract provided by the MKX Real Estate team. The main objective of the audit was to verify the security and functionality claims of the smart contract. The audit process involved a thorough manual code review and automated security testing.

Overall, the audit identified a total of 9 issue, classified as follows:

- 1 critical issue were found.
- 2 high severity issues were found.
- 3 medium severity issues were found.
- 1 low severity issues were discovered
- 2 informational issues were identified

The audit report provides detailed descriptions of each identified issue, including severity levels, proof of concepts and recommendations for mitigation. It also includes code snippets, where applicable, to demonstrate the issues and suggest possible fixes. We recommend the MKX Real Estate team to review the suggestions.

Update (01.07.2025): The MKX Real Estate team has successfully addressed all identified issues from the audit. All vulnerabilities have been mitigated based on the recommendations provided in the report. A follow-up review confirms that the fixes have been implemented effectively, ensuring the security and functionality of the smart contract. The updated codebase reflects the necessary improvements, and no further critical concerns remain.



8. About the Auditor

Established in 2017 under the name Chainsulting, and rebranded as softstack GmbH in 2023, softstack has been a trusted name in Web3 Security space. Within the rapidly growing Web3 industry, softstack provides a comprehensive range of offerings that include software development, cybersecurity, and consulting services. Softstack's competency extends across the security landscape of prominent blockchains like Solana, Tezos, TON, Ethereum and Polygon. The company is widely recognized for conducting thorough code audits aimed at mitigating risk and promoting transparency.

The firm's proficiency lies particularly in assessing and fortifying smart contracts of leading DeFi projects, a testament to their commitment to maintaining the integrity of these innovative financial platforms. To date, softstack plays a crucial role in safeguarding over \$100 billion worth of user funds in various DeFi protocols.

Underpinned by a team of industry veterans possessing robust technical knowledge in the Web3 domain, softstack offers industry-leading smart contract audit services. Committed to evolving with their clients' ever-changing business needs, softstack's approach is as dynamic and innovative as the industry it serves.

Check our website for further information: <https://softstack.io>

How We Work



1 -----

PREPARATION

Supply our team with audit ready code and additional materials



2 -----

COMMUNICATION

We setup a real-time communication tool of your choice or communicate via e-mails.



3 -----

AUDIT

We conduct the audit, suggesting fixes to all vulnerabilities and help you to improve.



4 -----

FIXES

Your development team applies fixes while consulting with our auditors on their safety.



5 -----

REPORT

We check the applied fixes and deliver a full report on all steps done.



hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984