ZEALYNX SECURITY

Web3 Security & Smart Contract Development

IPAL NETWORK

Security Assessment

July 31, 2025

Prepared by Zealynx Security
contact@zealynx.io
Zealynx.io

Sergio	@Seecoalba
Bloqarl	@TheBlockChainer

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Cyfrin, Monadex, Lido, Inverter, Ribbon Protocol, and Paragon.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent security assessment of the IPAL Network Smart Contracts was conducted by Zealynx Security, with a focus on the security aspects of the decentralized knowledge marketplace implementation.

We performed the security evaluation based on the agreed scope during 7 days, following our systematic approach and methodology. Based on the scope and our performed activities, our security assessment revealed 1 Critical, 3 High, 3 Medium, 9 Low and 2 Informational security issues.

3.2 About Ipal Network

IPAL is a decentralized protocol that revolutionizes knowledge exchange through blockchain technology. By creating NFT-gated access systems, IPAL provides knowledge creators with programmable tools to monetize their expertise without intermediaries. Users can subscribe to knowledge vaults by purchasing access NFTs that grant time-limited permissions to valuable content.

Built with upgradeable proxy patterns and co-ownership mechanisms, it enables collaborative content creation and fair revenue distribution. The protocol collects platform fees from subscription purchases to ensure sustainable operations and development.

Platform treasury mechanisms and upcoming $IPAL governance tokens incentivize ecosystem participation while supporting sustainable growth.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes 629 nSLOC - normalized source lines of code. The codebase implements an upgradeable proxy pattern utilizing standard security libraries, extends the ERC721 standard, and includes the $IPAL token contract built on the ERC20 standard.

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, **Formal Verification**, and **meticulous manual security review.**

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality**: We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices**: Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments**: We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over a 7-day engagement, the Zealynx Security team - including both senior auditors and interns - conducted a security audit of the IPAL Network Smart Contract implementation developed by the IPAL team.

The audit focused on identifying vulnerabilities, logic flaws, and implementation-level issues that could affect the decentralized knowledge marketplace's NFT-gated access system, revenue distribution, and platform security.

A total of 18 issues were identified and categorized as follows:
- 1 Critical severity
- 3 High severity
- 3 Medium severity
- 9 Low severity
- 2 Informational

The initial commit hashes audited are:
- 46672864c42fa379d32f5289bc992a8aa39c6944
- 80156f6ec201f23592089f7855e571b577666abc

The key findings include:

- **Missing access control in vault subscriptions:** The setSubscription function lacks ownership validation, allowing any address to create subscription offerings for vault IDs they don't own. This enables attackers to hijack vaults and make private content public without creator consent.
- **JSON injection vulnerability:** User-controlled strings in tokenURI are concatenated without sanitization, enabling metadata manipulation attacks that can deceive NFT buyers and cause marketplace integration issues.
- **Direct vault manipulation bypass:** The public setAccess function allows direct manipulation of access controls, bypassing business logic validation and creating state inconsistencies.
- **Treasury fund accountability gap:** The contract permits treasury transfers of 70% of token supply without any state tracking or transparency mechanisms, creating centralized control risks.

While the contract achieves its core knowledge marketplace functionality, critical improvements in access control validation, input sanitization, and treasury accountability are required to ensure platform security and maintain creator trust in the decentralized knowledge exchange ecosystem.

Summary of Findings :

Vulnerability	Severity
[C-01] Missing access control in setSubscription enables vault hijacking and revenue theft	**Critical**
[H-01] Missing JSON sanitization in tokenURI enables metadata injection attacks	**High**
[H-02] Public setAccess function bypasses business logic validation enabling direct vault manipulation	**High**
[H-03] Missing duplicate prevention in setSubscription allows multiple subscriptions per vault	**High**
[M-01] Missing token existence validation in tokenURI leads to EIP-721 standard violation and marketplace integration issues	**Medium**
[M-02] Missing state tracking in treasury transfers leads to unaccountable fund movements	**Medium**
[M-03] Transferable NFT design enables subscription sharing and secondary market revenue dilution	**Medium**
[L-01] Missing _disableInitializers() call in KnowledgeMargetV2 constructor	**Low**
[L-02] Missing Parent Contract Initialization in Upgradeable Contract	**Low**

Vulnerability	Severity
[L-03] Unofficial EIP-4908 designation creates trust and credibility concerns	Low
[L-04] Missing Storage Gaps in Upgradeable Contract	Low
[L-05] Missing active subscription validation in mint function leads to accidental duplicate purchases	Low
[L-06] Missing event emission in token distribution functions leads to lack of transparency and monitoring	Low
[L-07] Immutable platform fee configuration leads to inflexible fee structure and forced upgrades	Low
[L-08] 'SubscriptionCreated' Event Missing Co-ownership Parameters	Low
[L-09] Potential for Locked ETH Due to Direct Transfers	Low
[I-01] Inconsistent Use of Checks-Effects-Interactions (CEI) Pattern in _processPayment	Informational
[I-02] Unnecessary and Misleading check in implementation()	Informational

7. Audit Findings

7.1 Critical Severity Findings

[C-01] Missing access control in setSubscription enables vault hijacking and revenue theft

Description

The setSubscription function lacks access control validation, allowing any address to create subscription offerings for vault IDs they do not own. The function only validates input parameters but does not verify that msg.sender has legitimate ownership or control over the specified vaultId.

Vulnerable Scenario:
The following steps help understand the issue:

1. Alice creates a knowledge vault "alice-blockchain-course" on the IPAL platform and hosts content off-chain.
2. Bob monitors the platform and identifies Alice's popular vault ID.
3. Bob calls setSubscription("alice-blockchain-course", 0, 0, "", address(0), 0) directly on the smart contract.
4. Bob's subscription offering makes Alice's private vault public without her consent.
5. Users can now access Alice's premium content for free, bypassing her intended monetization.

Impact

- A non-owner could change the price or duration of a vault's subscription, potentially setting it to zero and making private content public without the owner's consent, or altering the terms of access.
- This would directly contradict the stated goal of the IPAL platform, which aims to create a decentralized marketplace where creators maintain control over their knowledge assets and enable content creators to monetize their knowledge.

Recommendation

1. Implement a vault ownership registry that maps vault IDs to their legitimate owners
2. Add access control validation in setSubscription to verify caller owns the vault

```
// Recommended fix
mapping(string => address) public vaultOwners;

function registerVault(string calldata vaultId) external {
    require(vaultOwners[vaultId] == address(0), "Vault already registered");
    vaultOwners[vaultId] = msg.sender;
}

function setSubscription(string calldata vaultId, ...) external {
    require(vaultOwners[vaultId] == msg.sender, "Not vault owner");
    // ... rest of function
}
```

Ipal Network:

Confirmed: The issue has been resolved by implementing a vault ownership registry and adding the necessary access control validation to the setSubscription function to prevent unauthorized modifications.

Zealynx:

Fixed: Added proper access control with vault ownership registry and validation in setSubscription function, preventing unauthorized vault hijacking.

7.2 High Severity Findings

[H-01] Missing JSON sanitization in tokenURI enables metadata injection attacks

Description

The tokenURI function constructs JSON metadata by directly concatenating user-controlled strings without proper sanitization, enabling JSON injection attacks. This vulnerability allows vault owners to inject malicious JSON syntax through the vaultId and imageURL parameters when calling setSubscription(), potentially manipulating the metadata structure and content displayed for their NFTs.

The vulnerable code directly embeds unsanitized user input into JSON strings:

```
string memory json = string.concat(
    "{",
        "\"name\":\"", data.resourceId, "\",",           //  Unsanitized user input
        "\"description\":\"This NFT grants access to a knowledge vault.\",",
        "\"external_url\":\"[https://knowledge-market.io/vaults/](https://knowledge-market.io/vaults/)",
        data.resourceId, "\",", //  Unsanitized user input
        "\"image\":\"", imageUrl, "\",",                 //  Unsanitized user input
        // ... rest of JSON construction
    "}"
);
```

Vulnerable Scenario:
The following steps help reproduce the issue:

1. A vault owner calls setSubscription() with malicious JSON characters in the vaultId parameter:
   'MyVault", "description": "HACKED!", "image": "ipfs://malicious.jpg", "name": "FakeNFT'
2. The system stores this malicious string without validation or sanitization.
3. When tokenURI() is called for an NFT from this vault, it constructs JSON by directly concatenating the malicious string.
4. The resulting JSON contains injected fields that can override intended metadata values:
   {"name": "MyVault", "description": "HACKED!", "image": "ipfs://malicious.jpg", "name": "FakeNFT", "description": "This NFT grants access to a knowledge vault.", ...}

5. JSON parsers typically use the last occurrence of duplicate keys, causing the injected values to override the legitimate ones.
6. Frontend applications and NFT marketplaces display the manipulated metadata to users.

Impact

Vault owners can manipulate the metadata display of their own NFTs through JSON injection, potentially deceiving buyers and causing integration issues. While the attack scope is limited to self-owned NFTs, it creates risks for marketplace integrity, user trust, and frontend security. The vulnerability can lead to display of misleading content, impersonation of valuable NFTs, and potential code execution if frontend applications use insecure JSON parsing methods like eval().

Recommendation

Implement the OWASP JSON Sanitizer library which is specifically designed to handle JSON injection vulnerabilities. This library converts JSON-like content to valid, secure JSON and is widely used in production systems.

Ipal Network:

Confirmed: Input sanitization has been added to the metadata construction process, preventing malicious data from being injected into the token URI.

Zealynx:

Fixed: JSON sanitization properly implemented with escape function for all user-controlled strings, plus Base64 encoding for additional security.

[H-02] Public setAccess function bypasses business logic validation enabling direct vault manipulation

Description

The setAccess function is declared as public, allowing any address to directly manipulate access control settings for any resource ID, bypassing the intended business logic and state management implemented in the higher-level KnowledgeMarketV2 contract.

Vulnerable Scenario:
The following steps help understand the issue:

1. Alice creates a premium knowledge vault "alice-course-123" and sets up a paid subscription through the platform.
2. Bob identifies Alice's vault ID and calls `setAccess("alice-course-123", 0, 0, address(0), 0)` directly on the ERC4908 contract.
3. Bob's call bypasses all validation and state management in setSubscription, making Alice's premium content free.
4. The platform's subscription tracking remains unchanged, creating inconsistency between business logic state and access control state.
5. Users can now mint access NFTs for free while the platform still shows Alice's original subscription settings.

Impact

Enables direct manipulation of vault access controls while bypassing business logic validation and state management, creating inconsistencies between the platform's subscription tracking and actual access permissions, and allowing attackers to undermine vault monetization without updating associated metadata.

Recommendation

Change the setAccess function visibility from public to internal to prevent direct external calls and force all access control modifications to go through the validated business logic layer:

```
function setAccess(
    string calldata resourceId,
    uint256 price,
    uint32 expirationDuration,
    address coOwner,
    uint32 splitFee
) internal {  // Changed from public to internal
    _setAccess(msg.sender, resourceId, price, expirationDuration, coOwner, splitFee);
}
```

Ipal Network:

Confirmed: We agreed with the recommendation and have updated the setAccess function's visibility from public to internal.

Zealynx:

Fixed: The setAccess function visibility changed from public to internal, preventing direct external manipulation and forcing all access control changes through proper business logic validation.

[H-03] Missing duplicate prevention in setSubscription allows multiple subscriptions per vault

Description

The setSubscription function lacks validation to prevent creating multiple subscription entries for the same vault ID. When called multiple times with the same vaultId, the function creates duplicate entries in the vaultOwnerSubscriptions array while overwriting the access control settings, violating the expected behavior that each vault should have only one subscription.

Vulnerable Scenario:
The following steps help understand the issue:

1. Alice calls setSubscription("vault-123", 100, 3600, ...) to create a subscription.
2. Alice later calls setSubscription("vault-123", 200, 7200, ...) with different parameters for the same vault.
3. The function creates a second entry in vaultOwnerSubscriptions[alice] with the same vault ID.
4. The underlying setAccess() overwrites the previous access control settings, making only the latest parameters active.
5. getVaultOwnerSubscriptions() returns duplicate entries for the same vault.
6. deleteSubscription() only removes the first matching entry, leaving orphaned duplicates.

Impact

Breaks the expected one-subscription-per-vault behavior, causing operational issues where vault owners can accidentally create multiple subscription entries for the same content, leading to UI confusion and incomplete deletion of subscription data.

Recommendation

Use a mapping to check for existing duplicates:

```
mapping(address ⇒ mapping(string ⇒ bool)) private hasSubscription;

function setSubscription(string calldata vaultId, ...) external {
    require(!hasSubscription[msg.sender][vaultId], "Subscription exists");
    hasSubscription[msg.sender][vaultId] = true;
    // ... rest of logic
}
```

Ipal Network:

Confirmed: An alternative solution has been implemented to prevent the creation of duplicate subscription entries for the same vault ID.

Zealynx:

Partially Fixed: Duplicate prevention implemented correctly, but the solution adds update logic to the create function instead of separating concerns into distinct setSubscription (create) and updateSubscription (update) functions, making the code harder to maintain and reason about.

UPDATE: Fixed

7.3 Medium Severity Findings

[M-01] Missing token existence validation in tokenURI leads to EIP-721 standard violation and marketplace integration issues

Description

The tokenURI function in both KnowledgeMarket.sol and KnowledgeMarketV2.sol does not validate that the requested tokenId exists before returning metadata, violating the EIP-721 standard. The function directly accesses storage mappings without checking token existence, causing it to return metadata with default/empty values for non-existent tokens instead of reverting as required by the standard.

```
function tokenURI(uint256 id) public view override returns (string memory) {
    Metadata memory data = nftData[id];  // No existence check
    Deal memory deal = dealInfo[id];
    Settings memory set = accessControl[data.hash];
    // ... returns metadata for non-existent tokens
}
```

EIP-721 Standard Requirement:
The EIP-721 specification explicitly states that tokenURI should "Throws if _tokenId is not a valid NFT."

Vulnerable Scenario:
The following steps help reproduce the issue:

1. An external application or user calls tokenURI(999999) for a token ID that was never minted.
2. The function accesses nftData[999999] which returns a default empty struct instead of reverting.
3. The function constructs and returns JSON metadata using these default values.
4. The caller receives seemingly valid metadata for a non-existent token, creating a "ghost token" effect.
5. NFT marketplaces and integrations may display this non-existent token as if it were real.

Impact

Direct violation of the EIP-721 standard leading to integration issues with NFT marketplaces and wallets. Applications may display non-existent tokens with default metadata, causing confusion for users and potentially breaking marketplace functionality that relies on standard-compliant behavior.

Recommendation

Add token existence validation at the beginning of the tokenURI function in both contracts:

```
require(_ownerOf(_tokenId) != address(0), "ERC721: invalid token ID");
```

```
function tokenURI(uint256 id) public view override returns (string memory) {
    _requireOwned(id);  // Add this line to validate token exists

    Metadata memory data = nftData[id];
    Deal memory deal = dealInfo[id];
    Settings memory set = accessControl[data.hash];

    // ... rest of function remains unchanged
}
```

Alternative if _requireOwned is not available:

```
function tokenURI(uint256 id) public view override returns (string memory) {
    ownerOf(id);  // This will revert if token doesn't exist

    Metadata memory data = nftData[id];
    // ... rest of function
}
```

Ipal Network:

Confirmed: We agreed with the recommendation and have implemented a token existence check at the beginning of the tokenURI function.

Zealynx:

Fixed: Added token existence validation with _requireOwned(id) at the beginning of tokenURI function, ensuring EIP-721 compliance and proper error handling for non-existent tokens.

[M-02] Missing state tracking in treasury transfers leads to unaccountable fund movements

Description

The transferFromTreasury function allows the owner to transfer tokens from the treasury address without any accounting or state tracking.
Unlike the distributeShareHolderTokens and airdrop functions which properly track allocations through state variables, treasury transfers are completely unaccounted for.

The contract allocates 70% of the total supply (700,000 tokens) to the treasury address during deployment, but provides no mechanism to track how these funds are used:

```
function transferFromTreasury(address to, uint256 amount) public onlyOwner {
    require(to != address(0), "Cannot transfer to zero address");
    _transfer(treasury, to, amount); // No state tracking
}
```

This creates an inconsistent accounting model where shareholder and airdrop allocations are carefully tracked, but the largest allocation (treasury) has no transparency or limits.

Vulnerable Scenario:
The following steps help understand the issue:

1. Contract deploys with 700,000 tokens allocated to treasury (70% of supply)
2. Owner can call transferFromTreasury to move any amount from treasury to any address
3. No state variables track how much treasury allocation has been used
4. No limits prevent owner from draining entire treasury allocation
5. Token holders have no visibility into treasury fund usage

Impact

Complete centralized control over 70% of token supply without transparency, accountability, or usage tracking.

Recommendation

Implement proper accounting for treasury transfers similar to other allocations:

```
uint256 private _unallocatedTreasuryTokens;

constructor(address _treasury) {
    // ... existing code ...

    // Track treasury allocation
    _unallocatedTreasuryTokens = treasuryShare;
    _mint(treasury, treasuryShare);
}

function transferFromTreasury(address to, uint256 amount) public onlyOwner {
    require(to != address(0), "Cannot transfer to zero address");
    if (amount > _unallocatedTreasuryTokens) {
        revert InsufficientTreasuryTokens();
    }
    _unallocatedTreasuryTokens -= amount;
    _transfer(treasury, to, amount);
}

function unallocatedTreasuryTokens() public view returns (uint256) {
    return _unallocatedTreasuryTokens;
}
```

Ipal Network:

Acknowledged: This finding relates to a contract that was decided to be outside the scope of the audit during the audit process.

Zealynx:

Acknowledged: The transferFromTreasury function still lacks state tracking for treasury fund usage, allowing unlimited transfers without accountability or transparency.

[M-03] Transferable NFT design enables subscription sharing and secondary market revenue dilution

Description

The access NFTs are fully transferable following standard ERC721 behavior, which enables subscription sharing and creates an uncontrolled secondary market that dilutes creator revenue. Once a user purchases access to a vault, they can freely transfer their NFT to others without any payment flowing back to the vault owner.

The system treats NFT ownership as the sole determinant of access rights, regardless of who originally paid for the subscription:

```
function hasAccess(
    address vaultOwner,
    string calldata vaultId,
    address consumer
) public view returns (bool response, string memory message, int32 expires) {
    // Only checks if consumer owns a valid NFT - not who paid for it
    for (uint256 i = 0; i < balanceOf(consumer); i++) {
        uint256 tokenId = tokenOfOwnerByIndex(consumer, i);
        // ... access granted based on ownership alone
    }
}
```

This creates a fundamental disconnect between payment and consumption, where vault owners receive one-time payments but potentially serve multiple users through NFT transfers.

Vulnerable Scenario:
The following steps help understand the issue:

1. Alice purchases a $100 NFT subscription for 30 days of vault access
2. Vault owner receives $100 payment and Alice gets the NFT
3. Alice consumes content for 10 days, then transfers NFT to Bob for $50 (external transaction)
4. Bob gains 20 days of access without any payment to the vault owner
5. Bob can further transfer to Charlie, and so on
6. Vault owner serves multiple users but only received one payment

Impact:

Revenue dilution for content creators, uncontrolled subscription sharing, and potential secondary market price competition that undermines the original pricing strategy..

Recommendations:

Consider implementing one of the following approaches based on business requirements:

Option 1: Non-transferable NFTs (Soulbound)

```
function _update(address to, uint256 tokenId, address auth)
    internal override returns (address) {
    address from = _ownerOf(tokenId);
    if (from != address(0) && to != address(0)) {
        revert TransfersNotAllowed();
    }
    return super._update(to, tokenId, auth);
}
```

Option 2: Transfer Restrictions with Grace Period

```
mapping(uint256 => uint256) public mintTimestamp;

function _update(address to, uint256 tokenId, address auth)
    internal override returns (address) {
    if (to != address(0) && mintTimestamp[tokenId] + LOCK_PERIOD > block.timestamp) {
        revert TransferLocked();
    }
    return super._update(to, tokenId, auth);
}
```

Ipal Network:

Confirmed: We have addressed this by implementing a one-day transfer lock on all newly minted access NFTs.

Zealynx:

Partially fixed: The NFTs remain fully transferable with no restrictions implemented. Users can still share subscriptions and create secondary markets, undermining the intended one-payment-per-user model and diluting creator revenue.

7.4 Low Severity Findings

[L-01] Missing _disableInitializers() call in KnowledgeMargetV2 constructor

Description

The implementation contract constructor lacks the _disableInitializers() call, allowing the implementation contract itself to be initialized directly. In upgradeable proxy patterns, the implementation contract should never be initialized - only the proxy should call the initialize() function.

Without _disableInitializers(), an attacker could potentially initialize the implementation contract directly, which could lead to unexpected behavior or serve as a stepping stone for more complex attacks.

The current constructor only calls the parent ERC4908 constructor but doesn't prevent initialization:

```
constructor() ERC4908("Knowledge Market Access", "KMA") {}
```

Recommendation

Add _disableInitializers() to the constructor to prevent direct initialization of the implementation contract:

```
constructor() ERC4908("Knowledge Market Access", "KMA") {
    _disableInitializers();
}
```

Ipal Network:

Confirmed: We agreed with the recommendation and have added the _disableInitializers() call to the constructor of our main implementation contract.

Zealynx:

Not Fixed: The actual implementation contract KnowledgeMarket.sol still lacks _disableInitializers() in its constructor. The fix exists only in the empty KnowledgeMarketV2.sol contract that has no business logic.

This architecture is flawed in design. The available options are:

- Add _disableInitializers() to the main KnowledgeMarket.sol contract, and delete KnowledgeMarketV2.sol
- Move all functionality to KnowledgeMarketV2.sol and use that as the implementation, and delete KnowledgeMarket.sol

UPDATE: Fixed

[L-02] Missing Parent Contract Initialization in Upgradeable Contract

Description

The contract is currently implemented as a non-upgradeable contract:
The contract uses non-upgradeable parent contracts (ReentrancyGuard) instead of their upgradeable versions (ReentrancyGuardUpgradeable) and fails to initialize parent contracts properly. This creates two issues:

1. Wrong contract type: Uses ReentrancyGuard which has a constructor that sets _status = _NOT_ENTERED in implementation contract storage, not proxy storage
2. Missing parent initialization: The initialize() function doesn't call parent initializers

Current problematic code:

```
// ❌ Line 6: Wrong import - should be ReentrancyGuardUpgradeable
import {ReentrancyGuard} from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";

// ❌ Line 13: Using non-upgradeable contract
contract KnowledgeMarket is Initializable, ERC4908, ReentrancyGuard {

// ❌ Lines 55-63: Missing parent initializations
function initialize(address payable _treasury, uint32 _fee) public initializer {
    if (_treasury == address(0)) revert ZeroAddress();
    if (_fee > 10000) revert InvalidFee();

    platformTreasury = _treasury;
    platformFeePercent = _fee;
    // Missing: __ERC721_init(), __ReentrancyGuard_init()
}
```

- ReentrancyGuard constructor sets _status = 1 in implementation contract storage
- In proxy pattern, this initialization happens in wrong storage context
- ReentrancyGuardUpgradeable requires explicit __ReentrancyGuard_init() call
- Without proper initialization, _status defaults to 0, but nonReentrant modifier checks _status != _ENTERED (2), so it still works by accident

Recommendations:

1. Change imports to upgradeable versions:

```
// ✅ Correct import
import {ReentrancyGuardUpgradeable} from
"@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol";
```

2. Update contract inheritance:

```
// ✅ Use upgradeable version
contract KnowledgeMarket is Initializable, ERC4908, ReentrancyGuardUpgradeable {
```

3. Initialize all parent contracts properly:

```
function initialize(address payable _treasury, uint32 _fee) public initializer {
    __ERC721_init("Knowledge Market Access", "KMA");
    __ERC721Enumerable_init();
    __ReentrancyGuard_init();

    if (_treasury == address(0)) revert ZeroAddress();
    if (_fee > 10000) revert InvalidFee();

    platformTreasury = _treasury;
    platformFeePercent = _fee;
}
```

Ipal Network:

Confirmed: We agreed with the recommendation.

Zealynx:

Not Fixed: The KnowledgeMarket.sol contract still lacks proper parent contract initialization. It should use ReentrancyGuardUpgradeable and call __ReentrancyGuard_init() in the initialize function.

The empty KnowledgeMarketV2.sol doesn't address this issue either. It doesn't even inherit from ReentrancyGuard at all, so it's not a proper replacement. To fix it, please follow the solution proposed in the Recommendations section.

UPDATE: Fixed

[L-03] Unofficial EIP-4908 designation creates trust and credibility concerns

Description

The protocol utilizes the designation "ERC-4908" for its token standard; however, this is not an official Ethereum Improvement Proposal (EIP) and does not appear in the official EIP repository or documentation. Using an unofficial EIP number creates several concerns:

It misleads users and developers into believing this is a standardized, peer-reviewed protocol when it is actually a custom implementation
It may cause when developers try to find official documentation or reference implementations
It can damage trust and credibility when users discover the EIP number doesn't correspond to any official standard, potentially making the platform appear deceptive or unprofessional.

The use of unofficial EIP numbers is particularly problematic in the blockchain space, where standards compliance and transparency are crucial for adoption and trust. Users and developers who perform due diligence by checking the official EIP documentation will find no reference to ERC-4908, which may raise concerns about the project's legitimacy and technical practices.

Recommendation

1. Remove the EIP designation and use a descriptive name for the contract (e.g., TimeBasedAccessNFT, SubscriptionAccessToken, or KnowledgeAccessNFT)
2. Update all documentation and references to reflect the custom nature of the implementation rather than implying it's a standard

```
// Instead of:
abstract contract ERC4908 is IERC4908, ERC721, ERC721Enumerable

// Use:
abstract contract TimeBasedAccessNFT is ITimeBasedAccessNFT, ERC721, ERC721Enumerable
// or
abstract contract SubscriptionAccessToken is ISubscriptionAccessToken, ERC721, ERC721Enumerable
```

Ipal Network:

Confirmed: We agreed with the recommendation and have renamed the contract to KnowledgeAccessNFT.

Zealynx:

Fixed: The contract no longer uses the unofficial "ERC4908" designation and has been renamed to KnowledgeAccessNFT.

[L-04] Missing Storage Gaps in Upgradeable Contract

Description

The upgradeable contracts lack storage gap variables (__gap). Storage gaps reserve storage slots for future versions, preventing storage collisions when new state variables are added during upgrades of parent contracts.

Recommendations:

Add a storage gap at the end of the contract to reserve slots for future upgrades:

```
// Reserved storage space to allow for layout changes in the future.
uint256[50] private __gap;
```

This should be placed just before the closing brace of the contract. The size of the gap (50 in this example) can be adjusted based on anticipated future needs, but 50 is a common value used in OpenZeppelin's upgradeable contracts.

Ipal Network:

Confirmed: We agreed with the recommendation.

Zealynx:

Not Fixed: Neither contract implements storage gaps. The main KnowledgeMarket.sol contract lacks the recommended uint256[50] private __gap; variable to reserve storage slots for future upgrades.

UPDATE: Fixed

[L-05] Missing active subscription validation in mint function leads to accidental duplicate purchases

Description

The mint function lacks validation to prevent users from purchasing multiple subscriptions for the same vault while they already have active access. This allows users to accidentally waste funds by purchasing redundant access to content they can already access.

The current implementation only checks if a subscription offering exists but does not verify whether the recipient already has active access to that vault:

```
function mint(
    address payable vaultOwner,
    string calldata vaultId,
    address to
) public payable nonReentrant {
    // ... validation checks ...

    bytes32 hash = _hash(vaultOwner, vaultId);
    if (!this.existAccess(hash)) revert MintUnavailable(hash);

    // Missing: Check if `to` already has active access

    // Always mints new NFT and charges user regardless of existing access
    _safeMint(to, tokenId);
}
```

This enables scenarios where users can accidentally purchase the same subscription multiple times, paying full price each time for access they already possess. This is particularly problematic in UI scenarios where users might not realize they already have active access or accidentally trigger multiple transactions.

Recommendations:

Add a validation check to prevent minting when the user already has active access to the vault:

```
function mint(
    address payable vaultOwner,
    string calldata vaultId,
    address to
) public payable nonReentrant {
    // ... existing validation checks ...

    bytes32 hash = _hash(vaultOwner, vaultId);
    if (!this.existAccess(hash)) revert MintUnavailable(hash);

    // NEW: Prevent minting if user already has active access
    (bool hasActiveAccess,,) = this.hasAccess(vaultOwner, vaultId, to);
    if (hasActiveAccess) revert AlreadyHasActiveAccess();

    // ... rest of minting logic ...
}
```

Additionally, implement a custom error:

```
error AlreadyHasActiveAccess();
```

Ipal Network:

Confirmed: We agreed with the recommendation and have added a validation check to the mint function.

Zealynx:

Fixed: Added active subscription validation in the mint function to prevent users from accidentally purchasing duplicate access to the same vault while they already have active access.

[L-06] Missing event emission in token distribution functions leads to lack of transparency and monitoring

Description

The token distribution functions distributeShareHolderTokens, airdrop, and transferFromTreasury do not emit events despite performing critical token allocation operations. These functions handle significant token movements (25% shareholder allocation, 5% airdrop allocation, and 70% treasury transfers) but provide no on-chain logging for transparency or off-chain monitoring.

```
function distributeShareHolderTokens(address to, uint256 amount) public onlyOwner {
    if (amount > _unallocatedShareHolderTokens) {
        revert InsufficientShareHolderTokens();
    }
    _unallocatedShareHolderTokens -= amount;
    _transfer(address(this), to, amount);
    // Missing event emission
}

function airdrop(address to, uint256 amount) public onlyOwner {
    if (amount > _unallocatedAirdropTokens) {
        revert InsufficientAirdropTokens();
    }
    _unallocatedAirdropTokens -= amount;
    _transfer(address(this), to, amount);
    // Missing event emission
}

function transferFromTreasury(address to, uint256 amount) public onlyOwner {
    require(to != address(0), "Cannot transfer to zero address");
    _transfer(treasury, to, amount);
    // Missing event emission
}
```

This lack of event emission prevents:

- Token holders from monitoring allocation distributions
- Off-chain applications from tracking token movements
- Analytics platforms from providing distribution insights
- Community governance from maintaining transparency
- Audit trails for token allocation decisions

Recommendations:

Add appropriate event emissions for all token distribution functions:

```
// Define events
event ShareHolderTokensDistributed(address indexed to, uint256 amount, uint256 remainingAllocation);
event AirdropExecuted(address indexed to, uint256 amount, uint256 remainingAllocation);
event TreasuryTransfer(address indexed to, uint256 amount);

function distributeShareHolderTokens(address to, uint256 amount) public onlyOwner {
    if (amount > _unallocatedShareHolderTokens) {
        revert InsufficientShareHolderTokens();
    }
    _unallocatedShareHolderTokens -= amount;
    _transfer(address(this), to, amount);

    emit ShareHolderTokensDistributed(to, amount, _unallocatedShareHolderTokens);
}

function airdrop(address to, uint256 amount) public onlyOwner {
    if (amount > _unallocatedAirdropTokens) {
        revert InsufficientAirdropTokens();
    }
    _unallocatedAirdropTokens -= amount;
    _transfer(address(this), to, amount);

    emit AirdropExecuted(to, amount, _unallocatedAirdropTokens);
}

function transferFromTreasury(address to, uint256 amount) public onlyOwner {
    require(to != address(0), "Cannot transfer to zero address");
    _transfer(treasury, to, amount);

    emit TreasuryTransfer(to, amount);
}
```

This would provide proper transparency and enable effective monitoring of token distribution activities.

Ipal Network:

Acknowledged: This finding relates to a contract that was decided to be outside the scope of the audit during the audit process.

Zealynx:

Acknowledged: The token distribution functions (distributeShareHolderTokens, airdrop, transferFromTreasury) still lack event emissions, preventing proper transparency and monitoring of token allocation activities.

[L-07] Immutable platform fee configuration leads to inflexible fee structure and forced upgrades

Description

The platform fee percentage can only be set once during contract initialization and cannot be updated afterward. The contract lacks any administrative function to modify the platformFeePercent value after deployment, making the fee structure permanently fixed.

```
function initialize(address payable _treasury, uint32 _fee) public initializer {
    if (_treasury == address(0)) revert ZeroAddress();
    if (_fee > 10000) revert InvalidFee();

    platformTreasury = _treasury;
    platformFeePercent = _fee; // Set once during initialization, no update mechanism
}
```

This creates several operational challenges:

- Market adaptation: Unable to adjust fees based on changing market conditions or user feedback
- Competitive response: Cannot respond to competitor pricing strategies or industry standards
- Revenue optimization: No ability to experiment with different fee structures to optimize platform revenue

The immutable fee structure may force the platform to choose suboptimal initial fees due to uncertainty about future market conditions, or alternatively require costly contract upgrades solely for fee adjustments.

Recommendations:

Implement an administrative function to update platform fees with appropriate access controls and safety measures:

```
// Add events for transparency
event PlatformFeeUpdated(uint32 oldFee, uint32 newFee);

// Add administrative function
function updatePlatformFee(uint32 _newFee) external onlyOwner {
    if (_newFee > 10000) revert InvalidFee();

    uint32 oldFee = platformFeePercent;
    platformFeePercent = _newFee;

    emit PlatformFeeUpdated(oldFee, _newFee);
}
```

Ipal Network:

Confirmed: We agreed with the recommendation.

Zealynx:

Not Fixed: The platform fee percentage can only be set once during contract initialization and cannot be updated afterward.

UPDATE: Fixed

[L-08] SubscriptionCreated Event Missing Co-ownership Parameters

Description

The SubscriptionCreated event does not emit the coOwner and splitFee parameters that are passed to the setSubscription function. This prevents off-chain services from tracking co-ownership details when subscriptions are created.

```
emit SubscriptionCreated(msg.sender, vaultId, price, expirationDuration);
```

Additionally, these parameters lack NatSpec documentation.
The function accepts coOwner and splitFee parameters but they are not included in the event emission or properly documented.

Off-chain services and indexers cannot track co-ownership details from events alone, requiring additional contract calls to retrieve this information. This reduces the utility of events for monitoring subscription creation and increases the complexity of integrating with the protocol.

Recommendations:

Update the SubscriptionCreated event definition and emission to include the missing parameters:

```
emit SubscriptionCreated(msg.sender, vaultId, price, expirationDuration, coOwner, splitFee);
```

Ipal Network:

Confirmed: We agreed with the recommendation, and the SubscriptionCreated and SubscriptionUpdated events have been updated to include the coOwner and splitFee parameters.

Zealynx:

Fixed: The SubscriptionCreated and SubscriptionUpdated events now properly include the coOwner and splitFee parameters in both their definitions and emissions, enabling off-chain services to track co-ownership details when subscriptions are created or updated.

[L-09] Potential for Locked ETH Due to Direct Transfers

Description

The KnowledgeMarket.sol contract handles ETH payments but lacks mechanisms to handle direct ETH transfers that might occur due to user error or misunderstanding of the payment process. Since the contract doesn't implement receive() or fallback() functions, any ETH sent directly to the contract address (outside of the normal mint() function) would become permanently locked.

Given that this contract deals with ETH payments, users might mistakenly send ETH directly to the contract address, expecting it to trigger some functionality, which would result in permanent loss of those funds.

Recommendations:

Implement a receive() function that reverts to prevent accidental direct transfers:

```
/**
 * @dev Prevents accidental direct ETH transfers
 * @notice Use the mint() function to purchase access tokens
 */
receive() external payable {
    revert("Direct ETH transfers not allowed. Use mint() function.");
}
```

Ipal Network:

Confirmed: We agreed with the recommendation and have implemented a receive() function that reverts on direct Ether transfers.

Zealynx:

Fixed: The contract now implements a receive() function that reverts with a clear error message "Direct ETH transfers not allowed. Use mint() function.", preventing accidental direct ETH transfers and potential permanent loss of funds.

7.5 Informational Findings

[I-01] Inconsistent Use of Checks-Effects-Interactions (CEI) Pattern in _processPayment

Description

In _processPayment, there's a CEI issue.
The current version updates state (remaining -= feeAmount) after the external call:

```
if (feeAmount > 0) {
    (bool sentFee, ) = platformTreasury.call{value: feeAmount}("");
    require(sentFee, "Failed to send platform fee");
    remaining -= feeAmount;
}
```

In the _processPayment function, the suggested solution is to move the remaining line -= feeAmount before the external call:

```
if (feeAmount > 0) {
+   remaining -= feeAmount;
    (bool sentFee, ) = platformTreasury.call{value: feeAmount}("");
    require(sentFee, "Failed to send platform fee");
-   remaining -= feeAmount;
}
```

Ipal Network:

Confirmed: We agreed with the recommendation to adhere strictly to the Checks-Effects-Interactions pattern.

Zealynx:

Not Fixed: The CEI pattern violation remains. State updates (remaining -= feeAmount) still occur after external calls, which goes against the recommended checks-effects-interactions pattern for security best practices.

UPDATE: Fixed

[I-02] Unnecessary and Misleading check in implementation()

Description

The proxy contract includes a misleading access control check in the implementation() function that serves no practical security purpose:

1. In KnowledgeMarketProxy.sol (line 54), the implementation address is restricted to admin-only access:

```
require(msg.sender == _getAdmin(), "Only admin can view implementation");
```

2. However, in ProxyAdmin.sol (line 47), the same implementation address is exposed publicly:

```
function getProxyImplementation(KnowledgeMarketProxy proxy) external view returns (address) {
    // We use the implementation getter directly on the proxy (no need to be admin)
    return proxy.implementation();
}
```

3. Additionally, the implementation address is stored in a standardized storage slot (EIP-1967) that anyone can read directly from the blockchain.

- This creates a false sense of security and architectural inconsistency. The access control in the proxy is ineffective since the same information is available through multiple public channels.

The recommended solution is to remove the admin-only check from the KnowledgeMarketProxy implementation() function:

```
    function implementation() external view returns (address) {
-       require(msg.sender == _getAdmin(), "Only admin can view implementation");
        return _getImplementation();
    }
```

Since the implementation address is publicly accessible through the ProxyAdmin contract and direct storage-slot queries, the check provides a false sense of security without offering any real protection.

Ipal Network:

Confirmed: We agreed with the recommendation.

Zealynx:

Not Fixed: The admin-only check in the implementation() function remains, but this is purely cosmetic since the implementation address is publicly readable through standard EIP-1967 storage slots anyway.

UPDATE: Fixed