# **WhitelabelNFT Smart Contract Report**

### **Date:** 24th March 2024

### **Contract Name:** WhitelabelNFT

### **Contract:** 0xd1EF6fAE85952F25D5C205c09389ea3692f4288F

### **Network:** Eth, Base

### **Overview**

This audit report presents an assessment of the WhitelabelNFT implementation Smart Contract to evaluate its security, reliability, and compliance with best practices. The audit aims to identify potential vulnerabilities, design flaws, and areas for improvement to enhance the overall security and functionality of the contract.

### 

### **Table of Contents**

| **Issue** | **Severity** |
| --- | --- |
| Late Verification of onERC721Received Function After Token Transfer | **Medium** |
| Missing Upper Bound Minting Limit | **Medium** |
| Lack of Null Check in setContractURI Function | **Medium** |
| Lack of Null Check in setContractURI Function | **Low** |
| Uninitialized State Variable in TinyERC721 Contract | **Low** |
| Lack of Zero-Check in RoyalERC721 Contract's transferOwnership Function | **Low** |
| Use of Block Timestamp in Time-based Comparisons in TinyERC721 Contract's Permit Functions | **Low** |
| Use of Assembly in AddressUtils, ArrayUtils, and ECDSA Libraries | **Low** |
| Incorrect Versions of Solidity Pragma Directives Used | **Low** |
| Usage of Literals with Too Many Digits | **Low** |

**Issue: Late Verification of onERC721Received Function After Token Transfer**

**Severity: Medium**

**Description:**

The safeTransferFrom, safeMint and safeMintBatch function in the TinyERC721 smart contract performs a verification check for the implementation of the onERC721Received function after the token transfer has been initiated using the transferFrom, mint and mintBatch function. This delayed verification introduces a potential risk where tokens may be transferred to a contract address that does not support the ERC721 interface, leading to loss of tokens or unexpected behavior.

### **Affected code and Elements** The affected code is the safeTransferFrom, safeMint and safeMintBatch function in TinyERC721 smart contract. This function is a critical component of the ERC721 token standard implementation. **Impact:**

* **Token Loss or Lock:** Delaying the verification of the recipient's ability to handle ERC721 tokens until after the token transfer increases the risk of tokens becoming inaccessible or permanently locked in contracts that do not implement the required onERC721Received function.
* **Smart Contract Vulnerabilities:** Interacting with contracts that do not properly handle incoming ERC721 tokens may result in vulnerabilities or unexpected behaviors, potentially compromising the security and integrity of the contract.

### **Recommendations:** To mitigate the risk associated with transferring tokens to contracts that do not support the ERC721 interface, it is recommended to perform the verification check for the implementation of the onERC721Received function before initiating the token transfer. By conducting this check before the transfer, the smart contract can ensure that tokens are only transferred to contracts capable of handling ERC721 tokens correctly, reducing the risk of token loss or contract vulnerabilities.

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### **Issue: Inconsistent Event Emission Timing in State-Modifying Functions**

**Severity: Medium**

**Description:**

The transferOwnership and setBaseURI functions in the smart contract emit events before updating the contract state. Emitting events before modifying the contract state may lead to inconsistencies and incorrect event logs if the state update fails or if the emitted events are not captured by listeners due to an error. This sequencing issue could result in misleading event logs or incorrect tracking of state changes.  
  
   
**Affected Code and Elements:**The affected code includes the transferOwnership and setBaseURI functions in the smart contract, where events (OwnerUpdated and BaseURIUpdated) are emitted before updating the contract state by modifying the owner and baseURI variables, respectively.

### **Impact:**

* **Inconsistent Event Logs:** Emitting the OwnerUpdated event before updating the contract state may lead to event logs that do not accurately reflect the actual ownership change.
* **Misleading Ownership Tracking:** If the state update fails after emitting the event, or if the event is not captured due to an error, the ownership change may not be properly tracked, leading to confusion or errors in ownership management.

### **Recommendations:**

To ensure consistency in event logging and accurate tracking of state changes, it is recommended to emit events after successfully updating the contract state. By emitting events after updating the contract state, the smart contract can provide reliable event logs that reflect the actual state changes and prevent misleading or inconsistent state tracking.  
  
**Reference:**

Solidity Documentation: <https://docs.soliditylang.org/en/v0.8.11/contracts.html#events>

**Issue: Missing Upper Bound Minting Limit**

**Severity: Medium**

**Description:**

The safeMintBatch and mintBatch functions lack an upper bound limit on the number of tokens that can be minted in a single transaction. Without an upper limit, an attacker could potentially exploit the contract to mint an excessive number of tokens in a single transaction, leading to unforeseen consequences such as overwhelming the contract state or causing denial-of-service (DoS) attacks.  
  
   
**Affected Code and Elements:**  
 - safeMintBatch Function (lines 1-26):

* + File: TinyERC721.sol
  + Lines: 1-26
* mintBatch Function (lines 29-70):
  + File: TinyERC721.sol
  + Lines: 29-70

### **Impact:**

The absence of an upper bound minting limit in the safeMintBatch and mintBatch functions poses a risk of potential abuse by malicious actors. Without proper constraints, attackers could exploit the contract to create an excessive number of tokens, causing congestion on the blockchain network, bloating the contract state, and potentially disrupting the normal operation of the application. Furthermore, excessive token minting could lead to unforeseen gas costs for users and contract administrators.

### **Recommendations:**

To mitigate the risk associated with unlimited token minting, consider implementing the following recommendations:

1. **Implement an Upper Bound Limit:** Introduce an upper bound limit on the number of tokens that can be minted in a single transaction. Define a reasonable threshold based on the expected usage of the application and the scalability requirements of the blockchain network.
2. **Use Safe Minting Practices:** Implement safe minting practices, such as rate limiting or batch processing, to prevent abuse and ensure the stability and reliability of the contract. Consider implementing mechanisms for verifying the legitimacy of minting requests and enforcing constraints on token creation.
3. **Audit and Test:** Conduct thorough audits and testing to identify and address potential vulnerabilities in the minting functionality. Test the contract under various conditions, including stress testing, to assess its resilience and scalability in handling token minting operations.

**Reference:**   
The issue of missing upper bound minting limits was identified through code analysis and review of the safeMintBatch and mintBatch functions in the TinyERC721.sol contract. Implementing an upper bound limit on token minting helps prevent abuse and ensures the stability and security of the contract. **Issue: Lack of Null Check in setContractURI Function**

**Severity: Low**

**Description:**

The setContractURI function in the smart contract updates the contract URI without performing a null check on the input parameter \_contractURI. This omission could potentially lead to unexpected behavior if the \_contractURI parameter is passed as null or an empty string. Without proper validation, setting the contract URI to null or an empty string may result in the loss of the contract's URI information, making it inaccessible or causing errors when accessed by users or applications.

### **Affected Code and Elements:**

The affected code is the setContractURI function in RoyalERC721 smart contract. This function is responsible for updating the contract URI, which is a crucial piece of metadata associated with the contract.

### **Impact:**

* **Loss of Contract URI:** Failing to perform a null check on the \_contractURI parameter allows setting the contract URI to null or an empty string, potentially leading to the loss of critical contract metadata.
* **Unexpected Behavior:** Setting the contract URI to null or an empty string may result in unexpected behavior when users or applications attempt to access or interact with the contract metadata, leading to confusion or errors in the system.

### **Recommendations:** To prevent the inadvertent loss of contract metadata and ensure consistent behavior, it is recommended to include a null check in the setContractURI function before updating the contract URI. By validating that the input parameter \_contractURI is not null or empty before performing the update, the smart contract can mitigate the risk of unintended consequences and maintain the integrity of its metadata. **Issue: Uninitialized State Variable in TinyERC721 Contract**

**Severity: Low**

**Description:**

The TinyERC721.collections state variable declared at line 138 in the TinyERC721 contract is never initialized. This uninitialized variable is referenced in various functions such as balanceOf, tokenOfOwnerByIndex, \_\_addLocal, \_\_addToken, \_\_addTokens, and \_\_removeLocal. This oversight may lead to unexpected behavior or runtime errors during contract execution.  
  
   
**Affected Code and Elements:**The affected code is in the TinyERC721 smart contract  
  
Variable → mapping(address => uint256[]) private collections;

Functions:-

* balanceOf(address)
* tokenOfOwnerByIndex(address,uint256)
* \_\_addLocal(uint256,address)
* \_\_addToken(uint256,address)
* \_\_addTokens(address,uint256,uint256)
* \_\_removeLocal(uint256)

### **Impact:**

The uninitialized state variable may cause incorrect data retrieval or manipulation, leading to contract malfunctions or vulnerabilities such as incorrect token balances or ownership tracking. This could result in user funds being locked or inaccessible.

### **Recommendations:**

Initialize the TinyERC721.collections state variable properly before using it in any functions. This can be accomplished by assigning an initial value to the variable during its declaration or initializing it in the contract constructor. Regular testing and auditing of the contract code should also be performed to ensure all state variables are properly initialized and used.  
**Reference:**   
The issue was detected using Slither, a Solidity static analysis framework. For more information on uninitialized state variables, refer to the [Slither Documentation on Uninitialized State Variables](https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables).

**Issue: Lack of Zero-Check in RoyalERC721 Contract's transferOwnership Function**

**Severity: Low**

**Description:**

The transferOwnership function in the RoyalERC721 contract fails to perform a zero-check on the \_owner parameter before assigning it to the owner state variable. This oversight could potentially allow setting the owner address to zero (address(0)), effectively transferring ownership of the contract to an invalid or non-existent address.

**Affected Contracts and Elements:**Affected Function: transferOwnership(address \_owner) in RoyalERC721.sol at line 187.

// RoyalERC721.sol - line 187

function transferOwnership(address \_owner) public {

// verify the access permission

require(isSenderInRole(ROLE\_OWNER\_MANAGER), "access denied");

// emit an event first - to log both old and new values

emit OwnerUpdated(msg.sender, owner, \_owner);

// emit zeppelin ownable-compliant ownership transfer event

emit OwnershipTransferred(owner, \_owner);

// update "owner" - Lacks zero-check

owner = \_owner;

}

### **Impact:**

The absence of a zero-check on the \_owner parameter could result in the contract's ownership being transferred to an invalid or non-existent address. This could potentially render the contract uncontrollable or inaccessible, leading to a loss of administrative control over the contract's functionalities.

### **Recommendations:**

To mitigate this issue, it is recommended to:

1. Implement a zero-check on the \_owner parameter before assigning it to the owner state variable.
2. Ensure that the new owner address is not set to zero (address(0)).
3. Conduct thorough testing and auditing of the contract to identify and address any similar oversight or vulnerability.

**Reference:**

The issue was identified through manual code review. It's important to ensure all state transitions, especially those involving critical contract functionalities like ownership transfer, are properly validated and safeguarded against unintended behavior or misuse.

**Issue Title:** **Use of Block Timestamp in Time-based Comparisons in TinyERC721 Contract's Permit Functions**

**Severity:** **Low**

**Description:**   
The permit and permitForAll functions in the TinyERC721 contract use the block timestamp (block.timestamp) for time-based comparisons. Relying solely on block timestamp for time-sensitive operations can introduce vulnerabilities due to potential manipulation of timestamps by miners or external attackers.

**Affected Code and Function:**

* Affected Functions:
  + permit(address, address, uint256, uint256, uint8, bytes32, bytes32) in TinyERC721.sol at lines 755-769.
  + permitForAll(address, address, bool, uint256, uint8, bytes32, bytes32) in TinyERC721.sol at lines 803-817.

**Solidity**

// TinyERC721.sol - line 765  
require(block.timestamp < \_exp, "signature expired");

// TinyERC721.sol - line 813  
require(block.timestamp < \_exp, "signature expired");

**Impact:**   
Relying solely on block timestamp for time-based comparisons can expose the contract to potential timestamp manipulation attacks, where attackers may attempt to manipulate the block timestamp to bypass time-based restrictions or exploit time-sensitive functionalities. This can lead to unauthorized access, front-running attacks, or other security vulnerabilities.

**Recommendation:**   
To mitigate this issue, it is recommended to:

1. Use block numbers or other mechanisms for time-based comparisons instead of relying solely on block timestamp.
2. Implement additional checks and validation mechanisms to ensure the integrity and security of time-sensitive operations.
3. Consider using external time oracles to obtain more reliable and tamper-proof timestamp data.

**Reference:**

The issue was identified using the Slither detector for dangerous block timestamp comparisons. It's essential to avoid relying solely on block timestamp for critical time-sensitive operations to enhance the security and robustness of the smart contract against potential vulnerabilities and attacks.

**Issue: Use of Assembly in AddressUtils, ArrayUtils, and ECDSA Libraries**

**Severity: Low**

**Description:**

The AddressUtils, ArrayUtils, and ECDSA libraries in the smart contract project utilize assembly language for certain operations. While assembly can provide optimizations and flexibility, it introduces complexity and potential security risks if not implemented carefully. Assembly code is less readable and harder to audit compared to Solidity code, making it more prone to errors and vulnerabilities.

### 

**Affected Contracts and Elements:   
 - AddressUtils.sol** - Inline assembly at lines 39-42:  
 // AddressUtils.sol - line 39-42  
 assembly {

result := gt(extcodesize(addr), 0)

}

- **ArrayUtils.sol** - Inline assembly at lines 35-97:  
 // ArrayUtils.sol - line 35-97

assembly {

// Assembly code for push32 function

}

**- ArrayUtils.sol** - Inline assembly at lines 44-48 and 53-58:  
 // ECDSA.sol - line 44-48 and 53-58

assembly {

// Assembly code for recover function

}

### 

### **Impact:**

Using assembly code increases the complexity of the contract and may introduce potential security vulnerabilities if not implemented correctly. Assembly language is more error-prone and harder to audit compared to Solidity, increasing the risk of introducing bugs or vulnerabilities that could be exploited by attackers.

### **Recommendations:**

To mitigate potential risks associated with assembly usage, consider the following recommendations:

1. **Thorough Testing:** Perform comprehensive testing, including unit tests and integration tests, to verify the correctness and security of assembly code.
2. **Code Review:** Conduct rigorous code reviews by experienced Solidity developers and security auditors to identify and address any vulnerabilities or potential issues in the assembly code.
3. **Documentation:** Provide detailed comments and documentation explaining the purpose and functionality of the assembly code to enhance readability and facilitate future audits and maintenance.
4. **Fallback to High-Level Constructs:** Whenever possible, prefer high-level Solidity constructs over assembly to improve code readability, maintainability, and security.

**Reference:**

The issue was identified using the Slither detector for assembly usage in smart contracts. Assembly code should be used judiciously and with caution, accompanied by thorough testing and review processes to ensure the security and reliability of the smart contract.

**Issue: Incorrect Versions of Solidity Pragma Directives Used**

**Severity: Low**

**Description:**

The project utilizes incorrect versions of Solidity pragma directives across its contracts, which may lead to potential issues such as outdated language features, deprecated functionalities, or known bugs associated with specific compiler versions. It's crucial to ensure that the chosen Solidity versions are appropriate for the project's requirements and align with best practices for smart contract development.  
  
  
**Affected Code and Elements:**  
 AccessControl.sol - Pragma version >=0.4.22

* EIP2981Spec.sol - Pragma version ^0.8.4
* ERC165Spec.sol - Pragma version ^0.8.4
* ERC721Spec.sol - Pragma version ^0.8.4
* ERC721SpecExt.sol - Pragma version ^0.8.4
* AddressUtils.sol - Pragma version ^0.8.4
* ArrayUtils.sol - Pragma version ^0.8.4
* ECDSA.sol - Pragma version ^0.8.4
* StringUtils.sol - Pragma version ^0.8.4
* RoyalERC721.sol - Pragma version ^0.8.4
* TinyERC721.sol - Pragma version ^0.8.4
* WhitelabelNFT.sol - Pragma version ^0.8.4

### **Impact:**

Using incorrect versions of Solidity pragma directives may result in various issues, including compatibility issues with compiler versions, potential vulnerabilities due to deprecated features or known bugs, and difficulties in maintaining and updating the codebase. Developers may encounter challenges in ensuring code correctness, reliability, and security when using incompatible or outdated Solidity versions.  
  
  
**Recommendations:**

To address this issue, consider the following recommendations:

1. Update Solidity Pragma Directives: Review and update the Solidity pragma directives in affected contracts to specify correct and compatible versions of the Solidity compiler. Ensure that the chosen versions are suitable for the project's requirements and align with recommended best practices for smart contract development.
2. Avoid Deprecated Compiler Versions: Avoid using deprecated or unsupported Solidity compiler versions, as they may contain known bugs or vulnerabilities that could impact the security and reliability of the smart contracts. Upgrade to newer and supported compiler versions to leverage the latest language features and optimizations.
3. Regularly Review and Update Pragma Directives: Establish a process for regularly reviewing and updating Solidity pragma directives to ensure that contracts remain compatible with the latest compiler versions and adhere to best practices for smart contract development. Monitor Solidity releases and announcements for any updates or changes that may affect contract compilation and deployment.

**Reference:**The issue was identified using the Slither detector for incorrect versions of Solidity pragma directives in smart contracts. Incorrect Solidity versions may introduce compatibility issues, vulnerabilities, and maintenance challenges, highlighting the importance of ensuring consistency and alignment with recommended best practices for smart contract development.

**Issue: Usage of Literals with Too Many Digits**

**Severity: Low**

**Description:**

The contracts utilize literals with an excessive number of digits, which may lead to readability issues and potential mistakes during code review or maintenance. While the code remains functional, readability is crucial for understanding and maintaining smart contracts over time. Using literals with a high number of digits can obscure the intent of the code and make it challenging to grasp the underlying logic at a glance.

**Affected Contracts and Elements:**TinyERC721.sol - Function: \_\_addLocal(uint256,address) (lines 1102-1114)

1. TinyERC721.sol - Function: \_\_removeLocal(uint256) (lines 1188-1222)
2. TinyERC721.sol - Function: \_\_removeToken(uint256) (lines 1236-1267)
3. WhitelabelNFT.sol - Function: slitherConstructorConstantVariables() (lines 17-28)

### **Impact:**

Using literals with too many digits may reduce code readability and increase the likelihood of errors or misunderstandings during code review, maintenance, or modification. Developers may find it challenging to comprehend the purpose and behavior of the code, especially when dealing with complex numerical values represented by lengthy literals. Additionally, excessive digit literals can make the code appear cluttered and less concise.

### **Recommendations:**

To address this issue, consider the following recommendations:

1. **Use Descriptive Constants:** Instead of relying on literals with numerous digits, define descriptive constants with meaningful names that convey the purpose and significance of the values. Constants enhance code readability and maintainability by providing context and clarity to numerical values used throughout the contract.
2. **Break Down Complex Expressions:** If the contract requires complex numerical expressions, break down the calculations into smaller, more manageable components with descriptive variable names. This approach improves code readability and comprehension while facilitating easier maintenance and debugging.
3. **Follow Coding Standards:** Adhere to coding standards and best practices that promote readability, such as limiting line length and avoiding excessively long literals. Consistent formatting and naming conventions contribute to code clarity and understanding, benefiting both developers and auditors.

**Reference:**

The issue was identified using the Slither detector for literals with too many digits in smart contracts. While not posing a critical risk, addressing this issue can enhance code readability and maintainability, contributing to the overall quality and reliability of the smart contract codebase.