V1 Migrator Audit Report

Date: January 2025
Prepared By: Owanemi Osaye-William

Table of Contents

Introduction

The purpose of this report is to document the findings from the security audit of the V1 Migrator smart contract. This audit was conducted to identify potential vulnerabilities and provide actionable recommendations to improve the contract's security and and adherance to best practices.

The findings are categorized with detailed descriptions, impacts, proof of concepts, recommended mitigations, and tools used. Individual bugs have been documented separately in the findings section.

Scope of Audit

The audit focused on the following aspects:

- Security vulnerabilities
- Code correctness and logic
- Adherence to best practices
- Gas efficiency

Methodology

The audit process involved:

- Manual code review
- Automated analysis using Slither
- Scenario-based testing using Foundry

Disclaimer

This report is based on the information provided at the time of the audit and does not guarantee the absence of future vulnerabilities. Subsequent security review and on-chain monitoring are strongly recommended.

Security Review Summary

review commit hash: be1b25ee800c99129d2c5390b1f5198986b2e0a0

Contract Scope The following smart contracts were in scope of the audit:

- `Interfaces/ICollectible.sol`
- `Interfaces/IERC21Receiver.sol`
- `COA-Contracts/Busd.sol`
- `COA-Contracts/Prlz.sol`
- `COA-Contracts/land-nfts/PLOT.sol`
- `COA-Contracts/land-nfts/ACRE.sol`
- `COA-Contracts/land-nfts/YARD.sol`
- `COA-Contracts/land-nfts-v2/PLOT.sol`
- `COA-Contracts/land-nfts-v2/ACRE.sol`
- `COA-Contracts/land-nfts-v2/YARD.sol`
- `Migrator.sol`

The following number of issues were found, categorized by their severity

- **Critical & High: 2 issues**
- **Medium: 2 issues**
- **Low: 10 issues**

Findings Summary

ID	Title	Severity
C-01	Incorrect token parameter transferred during ERC20 migration	Critical
H-01	Insecure payment validation in `_pay()` function	High
M-01	Potential Re-Entrancy vulnerability in all `mint()` functions in scoped contracts	Medium
M-02	Misconfigured authorization in `setFreeParticipant` function prevents controllers from managing free participants	Medium
L-01	Informational: Non-Standard Code Layout in all scoped contracts	Low
L-02	Informational: Poor Documentation/Natspec in all scoped Contracts	Low
L-03	Gas Optimization: Unnecessary storage reads from `_currentBatch` struct in all contracts scoped in `land-nfts` and `land-nfts-v2`	Low
L-04	Unused `_tax` Functionality in all contracts scoped in `land-nfts` and `land-nfts-v2`	Low

ID	Title	Severity
L-05	Missing zero address checks in all `land-nfts` and `land-nfts-v2` contracts	Low
L-06	Inconsistent error handling pattern in scoped contracts in `land-nfts-v2`	Low
L-07	Unused variables in `Migrator` contract	Low
L-08	Gas Optimization: make use of pre increment operator	Low
L-09	Code duplication in NFT migration logic	Low
L-10	Gas Optimization: Efficient function selector retrieval in `Migrator` contract	Low

Detailed Findings

[C-01] Incorrect amount parameter transferred during ERC20 migration

Description:

The `Migrator::migrateERC20Token` function transfers the wrong amount of tokenV2 to the user during an ERC20 token migration. Instead of transferring the calculated amount `tokenBToRecieve`, the function mistakenly transfers the input amount `_amount` of tokenV1. This results in users receiving fewer tokenV2 tokens than intended, violating the migration ratio defined by the price parameter. Additionally, the function updates a mapping `tokensMigrated[_token2] += tokenBToRecieve` with the correct `tokenBToRecieve` value, which demonstrates a mismatch between the transfer logic and the updated state. This further proves the existence of the bug and may mislead developers and users who rely on the data for verification.

Impact:

The incorrect transfer logic leads to users being under-compensated during migrations, undermining trust in the migration process and potentially resulting in financial loss for users. The mismatch between the updated ,mappings and the actual transfer amount highlights the issue, making it harder to detect and resolve discrepancies. If unaddressed, this bug could dissuade users from participating in token migrations and damage the credibility of the smart contract.

Proof of Concept:

Here is the scenario to demonstrate this:

Initial Balances:

- User A has 6 units of tokenV1 and 0 units of tokenV2. Migrator contract holds 1000 units of tokenV2.

Execution:

- User A calls migrateERC20Token(6, tokenV1, tokenV2) with a price of 2 set by the admin.

Expected Outcome:

- User A should receive 12 units of tokenV2. The event emitted: `TokenMigrationCompleted(userA, tokenv1, tokenV2, 6, 12)`.

Actual Outcome:

- User A receives only 6 units of tokenV2 due to the incorrect transfer logic. mapping updated misleads the user.

The following code segment in the migrateERC20Token function illustrates the bug:

```
success = IERC20Upgradeable(Requirements.tokenV2).transfer(
    _msgSender(),
@>    _amount // Incorrect: Should use tokenBToRecieve instead
);
```

When a user deposits 6 units of tokenV1 for migration, the expected behavior (assuming a price of 2) is that they receive 12 units of tokenV2. However, the actual behavior is that only 6 units of tokenV2 are transferred to the user. The mapping `tokensMigrated[_token2] += tokenBToRecieve` here correctly logs it;

Test Case to prove:

```
function testTokenBReceivedIsCorrectMigrationAmount() public {
    vm.startPrank(user);

    uint initialV1Balance = tokenV1.balanceOf(user);
    uint initialV2Balance = tokenV2.balanceOf(user);

    console.log("initialv1Balance: ", initialV1Balance);
    console.log("initialV2Balance: ", initialV2Balance);
    uint migrationAmount = 6; // Migrate 6 tokens

    // Approve migrator to spend tokens
    tokenV1.approve(address(migrator), migrationAmount);

    // Perform migration
    migrator.migrateERC20Token(
        migrationAmount,
        address(tokenV1),
        address(tokenV2)
    );

    // Check final balances
    uint finalV1Balance = tokenV1.balanceOf(user);
    uint finalV2Balance = tokenV2.balanceOf(user);
    console.log("finalv1balance: ", finalV1Balance);
    console.log("finalV2Balance: ", finalV2Balance);

    // Calculate expected amounts
    uint expectedV1Decrease = migrationAmount;
    uint expectedV2Increase = migrationAmount * 2; // Should get 2x tokens due to
price=2

    // Verify V1 token decrease is correct
    assertEq(
        initialV1Balance - finalV1Balance,
        expectedV1Decrease,
        "V1 token decrease incorrect"
    );

    // This assertion will fail because of the bug
    // User receives migrationAmount instead of migrationAmount * 2
    assertEq(
        finalV2Balance - initialV2Balance,
        expectedV2Increase,
        "V2 token increase incorrect - Bug detected!"
    );
    vm.stopPrank();
}
```

The output of the test is seen below:

```
$ forge test --match-test testTokenBReceivedIsCorrectMigrationAmount -vv
[⠊] Compiling...
[⠑] Compiling 3 files with Solc 0.8.27
[⠢] Solc 0.8.27 finished in 4.77s
Compiler run successful!

Ran 1 test for test/PoCs/MigratorPoc.sol:TokenBTransferAmountTest
[FAIL: V2 token increase incorrect - Bug detected!: 6 != 12]
test_tokenB_received_is_correct_migration_amount() (gas: 177947)
Logs:
  initialv1Balance:  6
  initialV2Balance:  0
  finalv1balance:  0
  finalV2Balance:  12

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 747.23ms (94.35ms
CPU time)

Ran 1 test suite in 1.13s (747.23ms CPU time): 0 tests passed, 1 failed, 0 skipped
(1 total tests)

Failing tests:
Encountered 1 failing test in test/PoCs/MigratorPoc.sol:TokenBTransferAmountTest
[FAIL: V2 token increase incorrect - Bug detected!: 4 != 8]
test_tokenB_received_is_correct_migration_amount() (gas: 177947)
```

it fails because the `_amount` the user inputted is being sent instead of `tokenBtoReceive`

The test confirms: The actual transfer is incorrect (6 units of tokenV2 instead of 12). The emitted event misleads by stating the correct amount (12).

Mitigation:

Replace the incorrect `_amount` variable with `tokenBToRecieve` in the transfer call:

```
success = IERC20Upgradeable(Requirements.tokenV2).transfer(
    _msgSender(),
-     _amount
);
```

```
success = IERC20Upgradeable(Requirements.tokenV2).transfer(
    _msgSender(),
+     tokenBToRecieve // Correct: Transfer the calculated tokenBToRecieve amount
);
```

Tools Used:

- Foundry
- Manual Review

[H-01] Insecure payment validation in `_pay()` function

Description

The `_pay()` function in all contracts scoped in `land-nfts` and `land-nfts-v2` always returns true regardless of the `transferFrom` result, rather than returning the result of the transfer. This creates an inconsistency in how payment success is reported

Although the `transferFrom` function of the token contract will revert the transaction if the transfer fails (due to insufficient funds, allowance issues, or other reasons), the `_pay()` function does not explicitly reflect this by returning the actual result of the transfer operation. Instead, it unconditionally returns true.

This is not a critical issue because a failed transfer with most ERC-20s will still halt execution, but it introduces ambiguity in the function's return value.

Proof of concept

If an ERC20 token that doesn't revert a transferFrom function is used as a payment token, the transfer `_pay()` would still return true an example of this is USDT which does not return a bool for certain methods

An extensive repo below shows ERC20 tokens with behaviour that may be surprising or unexpected WEIRD-ERC20 examples

Recommended Mitigation

Implement proper return value checking:

```
function _pay(address payee, uint256 quantity) internal virtual returns (bool) {
    IERC20Upgradeable token = IERC20Upgradeable(_paymentToken);
+    return token.transferFrom(payee, _feeCollector, _currentBatch.price *
quantity);
-    return true; // Always returns true regardless of transfer success
}
```

Tools used

- Manual Review
- Static analysis
- Slither

[M-01] Potential Re-Entrancy vulnerability in all `mint()` functions in scoped contracts

---

Description:

The `mint()` function in the scoped contracts in `land-nfts` and `land-nfts-v2` could potentially be vulnerable to re-entrancy attacks due to the `_pay()` function being called before the state variable `_currentBatch.quantity` is updated. Re-entrancy attacks can occur when external calls happen before the state changes are finalized.

While the function is currently safe from re-entrancy if no malicious token contract is used as the payment token, this safety depends entirely on the behavior of the payment token and its contract implementation. Malicious tokens could exploit this design by re-entering the `mint()` function during the transferFrom call in `_pay()`, leading to unintended behavior.

Impact:

If a malicious token is used as the payment token, it could execute a re-entrancy attack, repeatedly calling the `mint()` function before the state update `(_currentBatch.quantity = (_currentBatch.quantity - quantity))` is applied. This could result in unauthorized minting of tokens or exhaustion of the `_currentBatch.quantity`.

Recommended Mitigation:

Follow the CEI pattern (Checks-Effects-Interactions)

Which simply is,

1. making neccesary checks for subsequent actions. which was implemented correctly in the `mint()` fucntion

```
    function mint(uint256 quantity) public {
        if(_currentBatch.quantity <= 0){
            revert NoMoreTokensLeft();
        }
        if(!_currentBatch.active){
            revert CurrentBatchNotActive();
        }
        if(quantity <= 0){
          revert QuantityMustBeAboveZero();
        }

        if(quantity >= _maxBuyAmount && !hasRole(SIGNER_ROLE, _msgSender())){
            revert MaxBuyAmountLimitReached();
        }
```

2. then making state changes
3. and finally making external calls

below is the corrected version of the function

```
    function mint(uint256 quantity) public {
@>      //neccesary checks here
        if(_currentBatch.quantity <= 0){
            revert NoMoreTokensLeft();
        }
        if(!_currentBatch.active){
            revert CurrentBatchNotActive();
        }
        if(quantity <= 0){
          revert QuantityMustBeAboveZero();
        }

        if(quantity >= _maxBuyAmount && !hasRole(SIGNER_ROLE, _msgSender())){
            revert MaxBuyAmountLimitReached();
        }

@>        //perform state changes
+        _currentBatch.quantity = (_currentBatch.quantity - quantity);

@>        // carry out external call
        if (!freeParticipant[msg.sender]) {
            //require msg.sender is passed into _pay
            if(!_pay(msg.sender, quantity)){
                revert MustPayBeforeMinting();
            }
        }

        // check enough left to mint
-        _currentBatch.quantity = (_currentBatch.quantity - quantity);
        _safeMint(msg.sender, quantity);
    }
```

Tools used:

- Manual Review
- Static analysis
- Slither

[M-02] Misconfigured authorization in `setFreeParticipant` function prevents controllers from managing free participants

---

Description:

The `setFreeParticipant` function in all contracts scoped in `land-nfts` and `land-nfts-v2` iis ntended to allow both the owner and designated `freeParticipantControllers` to modify the `freeParticipant` mapping. However, the function is restricted by the onlyOwner modifier, allowing only the contract owner to execute it. This implementation prevents `freeParticipantControllers` from fulfilling their intended role, resulting in an ineffective delegation of authority.

Impact:

The current implementation creates the following issues: 1. Delegation Failure: The `freeParticipantControllers` mapping is non-functional because controllers cannot set free participants. 2. Unnecessary Gas Costs: Maintaining an ineffective `freeParticipantControllers` mapping introduces redundant storage costs.

Proof of Concept:

Current implementation:

```
function setFreeParticipant(address participant, bool free) public onlyOwner {
    freeParticipant[participant] = free;
}
```

Issue:

- The `onlyOwner` modifier restricts access to the owner only, making the `freeParticipantControllers` mapping unusable.

Expected Behavior:

- Both the owner and addresses in `freeParticipantControllers` should be able to execute the function.

Recommended Mitigation:

Update the `setFreeParticipant` function to allow both owner and free participant controllers to set free participants

```diff
- function setFreeParticipant(address participant, bool free) public onlyOwner {

+ function setFreeParticipant(address participant, bool free) external {
+        require(freeParticipantControllers[msg.sender] || msg.sender == owner(),
"Not authorized");
        freeParticipants[participant] = free;
}
```

Tools Used

- Manual Review
- Foundry

[L-01] - Informational: Non-Standard Code Layout in all scoped contracts

---

Description:

The contract does not follow Solidity's recommended code layout and organization practices, as outlined in the Solidity Style Guide. This deviation from best practices makes the code harder to read, understand, and maintain, especially as the contract evolves or expands.

Recommended Mitigation:

Contract should be layed out in the following order according to the official Solidity Style Guide

1. pragma
2. Import statements
3. Events
4. Errors
5. Interfaces
6. Libraries
7. Contracts

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State Variables

3. Events
4. Errors
5. Modifiers
6. External and Public view functions
7. Internal and private view functions

[L-02] - Informational: Poor Documentation/Natspec in all scoped Contracts

---

Description:

The follwing contracts lacks sufficient Documentation and NatSpec comments to explain the purpose and functionality of its components.

1. `src/COA-Contracts/land-nfts/PLOT.sol`
2. `src/COA-Contracts/land-nfts/ACRE.sol`
3. `src/COA-Contracts/land-nfts/YARD.sol`
4. `src/COA-Contracts/land-nfts-v2/ACRE.sol`
5. `src/COA-Contracts/land-nfts-v2/PLOT.sol`
6. `src/COA-Contracts/land-nfts-v2/YARD.sol`
7. `src/Migrator.sol`

Impact:

1. Increases the risk of misinterpretation during further development or auditing processes.
2. Makes integration with external systems and use by third parties less efficient due to lack of detailed documentation
3. Reduces code readability and maintainability.

[L-03] - Gas Optimization: Unnecessary storage reads from `_currentBatch` struct in all contracts scoped in `land-nfts` and `land-nfts-v2`

Description:

The `_currentBatch` struct is accessed directly from storage multiple times in the `mint` function in all contracts scoped in `land-nfts` and `land-nfts-v2`. This results in repeated and unnecessary storage reads, which are more expensive in terms of gas compared to memory access.

Impact:

Repeated storage reads increase the gas cost of the mint function and other functions that frequently access `_currentBatch`. This inefficiency can lead to higher transaction costs for users, particularly during minting events with high demand.

Recommended Mitigation:

Use a memory copy of the `_currentBatch` struct to reduce storage reads.

```
function mint(uint256 quantity) external {
+    Batch memory batch = _currentBatch;
-    require(_currentBatch.quantity > 0, "No more tokens left to mint");
-    require(_currentBatch.active, "Current Batch is not active");
+    require(batch.quantity > 0, "No tokens left in batch");
+    require(batch.active, "Batch is not active");
+    require(quantity > 0, "Quantity must be greater than zero");
+    require(quantity <= maxBuyAmount || msg.sender == owner(), "Exceeds max buy
limit");

    if (!freeParticipants[msg.sender]) {
        require(_pay(msg.sender, quantity), "Must Pay Minting fee");
    }

-   _currentBatch.quantity = (_currentBatch.quantity - quantity);
+    batch.quantity -= quantity;

+    // Write the updated memory struct back to storage
+    _currentBatch = batch;

    // Mint tokens
    _safeMint(msg.sender, quantity);
}
```

Tools used

- Manual Review
- Foundry

[L-04] Unused `_tax` Functionality in all contracts scoped in `land-nfts` and `land-nfts-v2`

Description:

all `land-nfts` and `land-nfts-v2` contracts includes redundant functionality in the form of the `_tax` function which is defined but never used in the contract. It performs a token transfer for a fixed transaction fee `_txFeeAmount` but does not serve any purpose in the current implementation.

Impact

Redundant functions increase the contract size unnecessarily, leading to higher gas costs for deployment and potential confusion for maintainers or auditors. They also create additional surface area for misuse or inconsistencies, especially during upgrades or audits, which could lead to security risks.

Proof of Concept:

This function is never called anywhere in the contract, making it dead code. It adds to the contract size without serving a purpose.

```
function _tax(address payee) internal virtual returns (bool) {
    IERC20 token = IERC20(_paymentToken);
    token.transferFrom(payee, _feeCollector, _txFeeAmount);
    return true;
}
```

Recommended Mitigation:

1. Remove the `_tax` function entirely unless there is a specific use case for it. the contract will be cleaner, more efficient, and easier to maintain.

```diff
-    function _tax(address payee) internal virtual returns (bool) {
-        IERC20 token = IERC20(_paymentToken);
-        token.transferFrom(payee, _feeCollector, _txFeeAmount);
-        return true;
-    }
```

Tools used:

- Manual Review

[L-05] Missing zero address checks in all `land-nfts` and `land-nfts-v2` contracts

---

Description:

the `setPaymentToken`, `setFeeCollector` `initilaze` fucntion in all `land-nfts` and `land-nfts-v2` contracts sets an address but doesn't validate for zero address

Recommended Mitigation:

Add zero address validation to `setPaymentToken`, `setFeeCollector` and `initilaze` functions in all scoped contracts in `land-nfts`and `land-nfts-v2`:

```diff
function setFeeCollector(address collector) public onlyOwner {
+   require(collector != address(0), "Invalid address");
    _feeCollector = collector;
}
```

Tools used:

- Manual Review
- Static analysis
- Slither

[L-06] Inconsistent error handling pattern in scoped contracts in `land-nfts-v2`

Description:

The land contracts in scope mixes different error handling patterns, using both custom errors without messages and TransactionFailed with string messages, leading to inconsistent gas consumption and potential confusion

Impact:

- Increased gas costs due to string storage in error messages
- Inconsistent error handling makes the contract harder to maintain
- Different gas costs for different error cases
-

Recommended Mitigation:

Tools used:

- Manual review

[L-07] Unused variables in `Migrator` contract

Description:

The following state variables are declared but never used in the `Migrator` contract:

- `totalAcreMigrated`
- `totalPlotMigrated`
- `totalYardMigrated`

This increases gas costs uneccesarily and introduces dead code

Impact:

Wasted storage allocation and increased gas costs.

Recommended Mitigation:

Remove the unused variables to save gas

```diff
- uint public totalAcreMigrated;
```

```diff
- uint public totalPlotMigrated;
```

```diff
- uint public totalYardMigrated;
```

Tools used:

Manual Review, Static analysis, Slither

[L-08] Gas Optimization: make use of pre increment operator

Description:

`Migrator` uses post-increment index++ in loops. Post-increment operators in solidity are less gas efficient as they require storing the original value.

Impact:

Higher gas costs in loops.

Recommended Mitigation:

Use pre-increment operators in loops:

```diff
- for (uint index = lastId; index < totalSupply; index++)
+ for (uint index = lastId; index < totalSupply; ++index)
```

Tools Used:

Manual Review

[L-09] Code duplication in NFT migration logic

Description:

`Migrator` contract duplicates NFT migration logic for each asset type (acre, plot, yard) as seen below

```
@> For Acre NFTs
if (_acre.length > 0) {
    for (uint i = 0; i < _acre.length; i++) {
        _withdrawOldNFT(Requirements.acre, _acre[i]);
    }
    _mintNewNFT(
        Requirements.acre,
        Requirements.acreV2,
        _msgSender(),
        _acre.length,
        _acre
    );
}

@> For Plot NFTs - Same logic repeated
if (_plot.length > 0) {
    for (uint i = 0; i < _plot.length; i++) {
        _withdrawOldNFT(Requirements.plot, _plot[i]);
    }
    _mintNewNFT(
        Requirements.plot,
        Requirements.plotV2,
        _msgSender(),
        _plot.length,
        _plot
    );
}

@> For Yard NFTs - Same logic repeated again
if (_yard.length > 0) {
    for (uint i = 0; i < _yard.length; i++) {
        _withdrawOldNFT(Requirements.yard, _yard[i]);
    }
    _mintNewNFT(
        Requirements.yard,
        Requirements.yardV2,
        _msgSender(),
        _yard.length,
        _yard
    );
}
```

Impact:

Increased deployment costs and reduced maintainability.

Recommended Mitigation:

Implement a shared migration function as done below:

create a new function `_migrateNFTBatch`

```
+   function _migrateNFTBatch(
+       address _nft1,
+       address _nft2,
+       uint[] memory _nfts
+   ) internal {
+       for (uint i = 0; i < _nfts.length; i++) {
+           _withdrawOldNFT(_nft1, _nfts[i]);
+       }
+       _mintNewNFT(_nft1, _nft2, _msgSender(), _nfts.length, _nfts);
+   }
```

the `migrateAllAsset` fucntion then becomes much cleaner:

```
function migrateAllAsset(
    uint[] memory _acre,
    uint[] memory _plot,
    uint[] memory _yard
) external returns (bool success) {
    uint migrateable = _acre.length + _plot.length + _yard.length;
    if (migrateable == 0) revert TransactionMessage("Not enough NFTs to migrate");

    if (_acre.length > 0) {
-           for (uint i = 0; i < _acre.length; i++) {
-               _withdrawOldNFT(Requirements.acre, _acre[i]);
-           }
-           _mintNewNFT(
-              Requirements.acre,
-               Requirements.acreV2,
-               _msgSender(),
-               _acre.length,
-               _acre
-           );
+        _migrateNFTBatch(Requirements.acre, Requirements.acreV2, _acre);
    }

    if (_plot.length > 0) {
-           for (uint i = 0; i < _acre.length; i++) {
-               _withdrawOldNFT(Requirements.acre, _acre[i]);
-           }
-          _mintNewNFT(
-               Requirements.acre,
-               Requirements.acreV2,
-               _msgSender(),
-               _acre.length,
-               _acre
-           );
+        _migrateNFTBatch(Requirements.plot, Requirements.plotV2, _plot);
    }

    if (_yard.length > 0) {
```

```
-             for (uint i = 0; i < _acre.length; i++) {
-                 _withdrawOldNFT(Requirements.acre, _acre[i]);
-             }
-             _mintNewNFT(
-                 Requirements.acre,
-                 Requirements.acreV2,
-                 _msgSender(),
-                 _acre.length,
-                _acre
-            );
+         _migrateNFTBatch(Requirements.yard, Requirements.yardV2, _yard);
    }

    return true;
}
```

[L-10] Gas Optimization: Efficient function selector retrieval in Migrator contract

Description

In the `onERC721Received` function in the migrator contract, the function selector is computed using a manual keccak256 hash of the function signature:

```
bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"));
```

This approach, while correct, is less gas-efficient than using the .selector property provided by Solidity. The .selector property generates the same result but is pre-determined at compile time, avoiding runtime computation and potential errors.

Impact

Using keccak256 to compute the selector at runtime increases gas consumption unnecessarily and introduces the possibility of typos or mismatches in the function signature, which may lead to unintended behavior.

Recommended Mitigation:

Use the built-in .selector property for retrieving the function selector, as it is more efficient and ensures correctness.

```
-   return bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"));
```

```
+    return this.onERC721Received.selector;
```

Tools used:

- Manual Reviw
- Foundry