Security Review Report
NM-0209-ZKVEGGIES

NETHERMIND
SECURITY

(April 03, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the smart contracts of zkVeggies. The audit scope comprises smart contracts for zkVeggies protocol, an NFT system that implements a unique "caps" mechanism for minting.

Any user can mint tokens under the free tier. There is a limit to the maximum number of tokens a user can hold at any point in time. Users can mint up to a max of 3 tokens in one cycle. There is a cooling period between cycles.

Users can also opt for paid minting. There are normal and premium cap plans to buy credits for minting. There is also a premium pack that entitles users with a defined set of normal and premium caps.

The minted NFT tokens have a royalty support and lock mechanism. Owners are allowed to burn their NFT tokens.

This security review focuses exclusively on the smart contracts listed in Section 2 (*Audited Files*).

The audited code comprises of 324 lines of code written in the Solidity language, and the audit was performed using (a) manual analysis of the code base and (b) creation of test cases. **In this document, we report** 11 points of attention, where none are classified as `Critical`, two is classified as `High`, one is classified as `Medium`, two are classified as `Low`, and six are classified as `Informational` and `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 presents the additional observations made during the audit. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, and automated tests. Section 10 concludes the document.

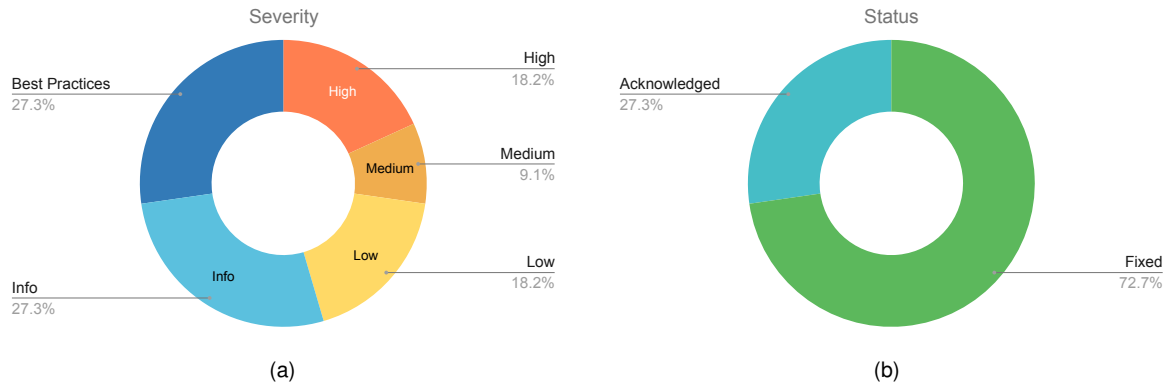

Fig. 1: Distribution of issues: Critical (0), **High** (2), **Medium** (1), **Low** (2), **Undetermined** (0), **Informational** (3), **Best Practices** (3). **Distribution of status: Fixed** (8), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan 31, 2025
Response from Client	Regular responses during audit engagement
Final Report	April 03, 2025
Repository	zkVeggies
Commit (Audit)	d44fc8e4c91ea8e209a915a59f5bce4a741214ee
Commit (Final)	1ea7178c14d394194c19f79af5b545b5f15c5a3f
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	ERC721TransferLock.sol	18	12	66.7%	6	36
2	VeggiaERC721.sol	260	290	111.5%	88	638
3	proxy/VeggiaERC721Proxy.sol	46	32	69.6%	13	91
	Total	**324**	**334**	**103.1%**	**107**	**765**

3 Summary of Issues

	Finding	Severity	Update
1	Name and Symbol for VeggiaERC721 are not set	High	Fixed
2	`VeggiaERC721` is vulnerable to signature replay attacks	High	Fixed
3	Tokens minted by smart contract could be stuck	Medium	Fixed
4	Storage collision in the Proxy contract	Low	Fixed
5	Users can get immediately all the free mint tokens without waiting post contract deployment	Low	Acknowledged
6	Missing validation in `setFreeMintCooldown` function could lead to revert	Info	Fixed
7	Caps purchased with `premiumPackPrice` is inconsistent with comments in the contract	Info	Fixed
8	`CapsPrice` should be configurable only for multiples of 3	Info	Fixed
9	Ownership is more secure with two-step transfers	Best Practices	Fixed
10	constructor in `VeggiaERC721` contract is redundant	Best Practices	Acknowledged
11	Unused custom error in `VeggiaERC721` contract	Best Practices	Acknowledged

4 System Overview

zkVeggies protocol is an NFT-based system that leverages upgradeable proxy patterns to extend functionality with new deployment. It allows the users to mint NFT based on their entitlements. Every user is entitled to mint up to a maximum limit of holding at a given point. The system also enforces cooldown time down between mints for the free tier. Users can also purchase entitlements to mint tokens. zKVeggies supports normal caps, premium caps, and Premium packs as plans for buying entitlements to mint NFTs.

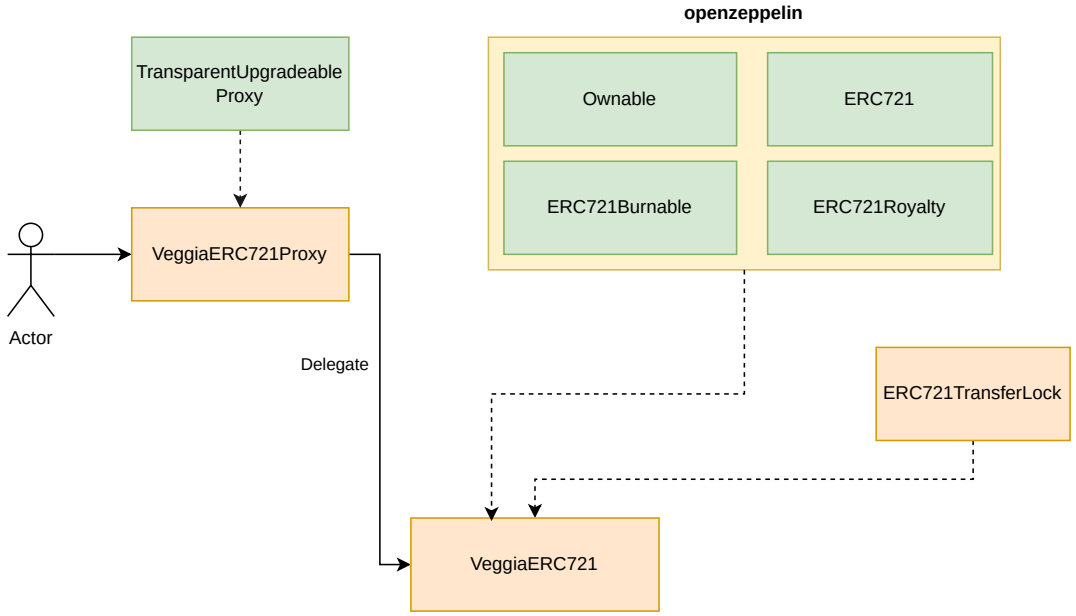

Fig. 2: zkVeggies overview

4.1 Free Minting

Anyone can mint tokens under the free tier. Users can mint NFTs using free caps that regenerate over time, limited by a cooldown period and maximum cap limit.

```
function freeMint3() external {
  ...
}
```

4.2 Paid Minting

Users can purchase normal and premium caps to mint NFTs. A special bundle containing 1 NFT, 12 regular caps, and 3 premium caps is also an option.

- buyCaps: Users can buy either normal or premium caps using this method. Users can only buy in the multiples of 3. The choice for normal or premium caps is based on the isPremium flag. The funds are paid through native tokens. This function adds entitlements to mint.

  ```
  function buyCaps(bool isPremium, uint256 quantity) external payable {
    ...
  }
  ```

- buyPremiumPack: Using this function, the user can purchase the special bundle that contains 1 NFT, 12 regular caps, and 3 premium. This function adds entitlements to mint. caps

  ```
  function buyPremiumPack() external payable {
   ...
  }
  ```

- mint3: User can mint their entitled tokens with this function.

```
  function mint3(bool isPremium) external {
   ...
}
```

4.3 Signature base minting

Users can claim tokens if they are granted entitlements through a signed message.

```
function mint3WithSignature(bytes memory signature, bytes calldata message) external {
 ...
}
```

4.4 Owner functions

The owner of the protocol can perform admin functions to configure the values in the protocol.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Name and Symbol for VeggiaERC721 are not set

File(s): VeggiaERC721.sol

Description: VeggiaERC721 is implemented behind proxy contract. All the interactions of the functionality in VeggiaERC721 are routed through the proxy contract. As a result, the storage used by the protocol is that of a proxy contract.

In the constructor, the name and symbol for the contract as passed as part of the inheritance hierarchy. The constructor is only initializing the storage of VeggiaERC721 contract, which is a logic contract.

```
constructor(address _feeReceiver, string memory _baseUri) ERC721("Veggia", "VEGGIA") Ownable(msg.sender) {
        baseURI = _baseUri;
        feeReceiver = _feeReceiver;
    }
```

The actual storage is owned by proxy and hence name and symbol should be initialized in the initialize function. However, the latter does not initialise the name and the symbol:

```
function initialize(address _owner, address _feeReceiver, address _capsSigner, string memory _baseUri) external {
    /// @dev Skips owner verification as the proxy is already ownable.
    /// @dev Skips initialization check as the proxy handles the initialization check internally.

    _transferOwnership(_owner);

    baseURI = _baseUri;
    capsSigner = _capsSigner;
    feeReceiver = _feeReceiver;

    // Must be a multiple of 3
    freeMintLimit = 6;
    freeMintCooldown = 12 hours;

    // Prices
    capsPriceByQuantity[3] = 0.0003 ether;
    capsPriceByQuantity[9] = 0.0006 ether;
    capsPriceByQuantity[30] = 0.0018 ether;
    premiumCapsPriceByQuantity[3] = 0.0009 ether;
    premiumCapsPriceByQuantity[9] = 0.00225 ether;
    premiumCapsPriceByQuantity[30] = 0.0054 ether;
    premiumPackPrice = 0.0036 ether;

    // Set the default royalty to 0
    _setDefaultRoyalty(feeReceiver, 0);
}
```

Moreover, ERC721 does not provide an option to update these variables through set functions.

As a result, the name and symbol are never initialised for VeggiaERC721

Recommendation(s): Use upgradeable contracts that are designed for proxy patterns.

Status: Fixed

Update from the client:commit 719b7b

6.2 [High] `VeggiaERC721` is vulnerable to signature replay attacks

File(s): `VeggiaERC721.sol`

Description: The VeggiaERC721 contract supports offchain signatures for minting tokens. The `mint3WithSignature` function enables users to mint tokens using off-chain signature. However, the current implementation is vulnerable to replay attack if `VeggiaERC721` is deployed on multiple chains.

```
function mint3WithSignature(bytes memory signature, bytes calldata message) external {
    // Verify the signature
    bytes32 messageHash = keccak256(message);
    address recoveredSigner = ECDSA.recover(messageHash, signature);
    if (recoveredSigner != capsSigner) revert INVALID_SIGNATURE();

    // Decode the message
    (address to, uint256 index, bool isPremium) = abi.decode(message, (address, uint256, bool));

    // Check if the signature has already been used
    if (signatureMintsOf[to][index]) revert SIGNATURE_REUSED();

    // Check if the sender is the expected one
    /// @dev Only the "to" address can use the signature
    if (to != msg.sender) revert INVALID_SENDER(msg.sender, to);

    // Mark the signature as used
    signatureMintsOf[to][index] = true;

    // Mint the NFTs
    _open3CapsForSender(isPremium);

    emit MintedWithSignature(msg.sender, message, signature);
}
```

The current implementation provides a guard when `VeggiaERC721` is deployed on a single chain. But, in case the contracts are deployed on multiple chains, using the same signature, the tokens can be minted on multiple chains.

Recommendation(s): Extend `VeggiaERC721` from `EIP712` which will ensure signatures cannot be replayed across chains. In order to be effective cross-chain, the correct domain initialization for `VeggiaERC721` should be implemented.

Status: Fixed

Update from the client: commit 6eab85

6.3 [Medium] Tokens minted by smart contract could be stuck

File(s): `VeggiaERC721.sol`

Description: Minting ERC721 tokens from a smart contract has a risk of tokens being stuck, in case the receiving contract does not have the ability to handle ERC721 tokens. Tokens locked in such contracts are not retrievable.

In order to prevent this, the recommended approach is to mint using _safeMint function. This function invokes hooks on the receiver to ensure the receiver can handle NFT tokens. It will proceed with the minting only for smart contracts that handle NFT tokens and hence prevent the tokens from being locked.

```
  function _open3CapsForSender(bool isPremium) private {
        //...
        uint256 _tokenId = tokenId;

        _mint(msg.sender, _tokenId);
        _mint(msg.sender, _tokenId + 1);
        _mint(msg.sender, _tokenId + 2);
       //...
    }
```

Note: While `_safeMint` will prevent locking of the ERC721 token in a smart contract, it invokes a hook `onERC721Received` which can expose to reentrancy. So, when switching to `_safeMint` reentrancy guard should be applied.

Recommendation(s): Consider using `_safeMint` function with reentrancy guard.

Status: Fixed

Update from the client: Commit 020286a

6.4 [Low] Storage collision in the Proxy contract

File(s): VeggiaERC721.sol

Description: In proxy-based contract architectures, maintaining a correct storage layout is crucial for ensuring proper contract functionality. When using a sequential storage layout, storage slots are assigned in a rigid and incremental manner, which can lead to unintended storage overwrites between the proxy and the implementation contract.

In the `VeggiaERC721Proxy` contract, the `initialized` state variable is stored in slot 0:

```
contract VeggiaERC721Proxy is TransparentUpgradeableProxy {
    /// @dev The initialization status of the contract.
    bool initialized;
    // --SNIP
}
```

Meanwhile, the implementation contract, `VeggiaERC721`, follows a sequential storage layout and inherits from `ERC721`, where the first storage slot is used for storing the NFT's `name`:

```
contract VeggiaERC721 is ERC721, ERC721Burnable, ERC721TransferLock, ERC721Royalty, Ownable {}

abstract contract ERC721 is Context, ERC165, IERC721, IERC721Metadata, IERC721Errors {
    using Strings for uint256;
    string private _name;
    // --SNIP
}
```

Due to this layout mismatch, when the `VeggiaERC721` contract initializes the `_name` variable, it writes to storage slot 0, which is already occupied by the initialized variable in the proxy contract. Consequently, this results in a direct override of the proxy contract's initialized state.

Fortunately, since the `_name` variable's byte representation will always be greater than `0x0`, the `initialized` flag will not be unintentionally reset to `false`, preventing a critical oversight where the contract can be reinitialized. However, storage collisions like this are serious concerns as they may lead to unexpected behaviors in future upgrades. Moreover, if new state variables are introduced in the future, they may cause additional conflicts unless proper storage gaps are allocated in advance.

Recommendation(s): To prevent storage collisions, it is strongly recommended to use OpenZeppelin's upgradable contract libraries. These libraries manage storage layouts explicitly and ensure that new storage variables can be safely added without causing conflicts. Additionally, when designing proxy-based contracts, explicitly reserving storage gaps (__gap patterns) can help accommodate future variables

Status: Fixed

Update from the client:Commit 0298d2e

6.5 [Low] Users can get immediately all the free mint tokens without waiting for post contract deployment

File(s): VeggiaERC721.sol

Description: In VeggiaERC721 contract, the freeMint3 function allows the user to mint free tokens up to a limit. After they are minted, there is a cool down period which they have to wait before the next mint.

However, the current implementation of freeMint3 function allows the users to claim their first 6 caps without any wait post-contract deployment.

```
function freeMint3() external {
    uint256 _freeMintLimit = freeMintLimit;
    uint256 _freeMintCooldown = freeMintCooldown;

    // Time elapsed since the last mint
==> uint256 elapsedTime = block.timestamp - lastMintTimestamp[msg.sender];

    uint256 intervals = elapsedTime / _freeMintCooldown;

    // Total accumulated caps balance
    uint256 totalCaps = intervals * 3;
    if (totalCaps > _freeMintLimit) {
        totalCaps = _freeMintLimit;
    }

    //...
}
```

The logic calculates the elapsed time as the difference between the current time and lastMintTimestamp of the users. As the default value for any account in lastMintTimestamp will be zero, every user will be able to claim their first set of free tokens without any wait for post-contract deployment.

Recommendation(s): One possible recommendation is to introduce a variable that tracks the time of the contract's deployment (set in the initialize function). During the first user's mint, consider calculating the elapsed time using the contract's deployment timestamp instead of lastMintTimestamp:

```
   uint256 elapsedTime = lastMintTimestamp[msg.sender] == 0 ?
                block.timestamp - deployedTimestamp :
                block.timestamp - lastMintTimestamp[msg.sender];
```

Status: Acknowledged

Update from the client: It is a design choice to let the user mint tokens immediately the first time.

6.6 [Info] Missing validation in `setFreeMintCooldown` function could lead to revert

File(s): VeggiaERC721.sol

Description: In `setFreeMintCooldown` function, there is no validation for an acceptable range of values for cool down time to unlock a new freeMint3. As a result, if it was set to `0`, it will cause a revert during `freeMint3` function call.

```
function freeMint3() external {
      //...
     uint256 _freeMintCooldown = freeMintCooldown;

     //...
    uint256 intervals = elapsedTime / _freeMintCooldown;

     //...
}
```

Likewise, many of the owner functions do not validate input values. Below are a few examples of such functions.

```
function setFreeMintCooldown(uint256 cooldown) external onlyOwner {
  freeMintCooldown = cooldown;  // can be set to 0
}
function setFeeReceiver(address receiver) external onlyOwner {
 feeReceiver = receiver; // can be set to address(0)
}
function setCapsSigner(address _capsSigner) external onlyOwner {
 capsSigner = _capsSigner; // can be set to address(0)
}
```

Recommendation(s): Add validation that restricts acceptable range of values for `freeMintCooldown` state variable. Also, add validations to owner functions.

Status: Fixed

Update from the client:Commit f9143a28

6.7 [Info] Caps purchased with `premiumPackPrice` is inconsistent with comments in the contract

File(s): VeggiaERC721.sol

Description: The caps purchased with `premiumPackPrice` have two different configurations per the comments in the contract. At the time of declaration of the state variable, the pack contains 1 NFT, 10 caps, and 3 premium caps.

```
* @notice The price of the premium pack.
    *         1 NFT mint + 10 caps + 3 premium caps.
    */
uint256 public premiumPackPrice;
```

But `buyPremiumPack` function mints 1 NFT, 12 caps and 3 premium caps.

```
/**
    * @notice Buy a premium pack with the price of {premiumPackPrice}.
    * @dev The premium pack contains 1 NFT mint + 12 caps + 3 premium caps.
    */
function buyPremiumPack() external payable {
    if (msg.value != premiumPackPrice) revert WRONG_VALUE();

    // Give the caps to the buyer (12 caps because 12 is a multiple of 3)
    paidCapsBalanceOf[msg.sender] += 12;
    // Give the premium caps to the buyer
    paidPremiumCapsBalanceOf[msg.sender] += 3;

    // Mint the NFT in the pack
    uint256 _tokenId = tokenId; // Cache the tokenId in memory
    _mint(msg.sender, _tokenId);

    //....
}
```

Recommendation(s): Update the natspec to reflect the actual caps minted for a user in a premium pack.

Status: Fixed

Update from the client:Commit 719b7b4

6.8 [Info] `CapsPrice` should be configurable only for multiples of 3

File(s): VeggiaERC721.sol

Description: Since the `buyCaps` function only allows buying caps in multiples of 3, the configurable prices should also be restricted to multiples of 3.

Currently, there is no validation in the set functions:

```
function setCapsPrice(uint256 quantity, uint256 price) external onlyOwner {
    capsPriceByQuantity[quantity] = price;
    emit CapsPriceChanged(quantity, price);
}

function setPremiumCapsPrice(uint256 quantity, uint256 price) external onlyOwner {
    premiumCapsPriceByQuantity[quantity] = price;
    emit PremiumCapsPriceChanged(quantity, price);
}
```

So, the owner can potentially configure any quantity, but the user can only buy in multiples of 3.

```
function buyCaps(bool isPremium, uint256 quantity) external payable {
    if (quantity % 3 != 0) revert WRONG_CAPS_QUANTITY();

    //...
}
```

Recommendation(s): Implement validation similar to the `buyCaps` function in the `setCapsPrice` and `setPremiumCapsPrice` functions.

Status: Fixed

Update from the client: Commit 8e71f782

6.9 [Best Practices] Ownership is more secure with two-step transfers

File(s): VeggiaERC721.sol

Description: As the owner is a critical role in managing admin functions of the protocol, it is recommended to implement a two-step transfer process. In the first step, the current owner initiates the process for the new account to take over ownership. As a second step, the new account will claim ownership. This ensures that the caller of the claiming function has control of the new account.

```
contract VeggiaERC721 is ERC721, ERC721Burnable, ERC721TransferLock, ERC721Royalty, Ownable {
```

Recommendation(s): Consider using `Ownable2Step` library contract of openzeppelin instead of `Ownable`.

Status: Fixed

Update from the client: Commit 0c1b5634

6.10 [Best Practices] The constructor in `VeggiaERC721` contract is redundant

File(s): VeggiaERC721.sol

Description: The implementation of the constructor in `VeggiaERC721` is incorrect and redundant. For the proxy pattern, the storage variable should be initialized only via the `initialize` function so that the storage in the proxy is set.

The below implementation of the constructor initializes the storage on the logic contract, which will not be used by the protocol.

```
constructor(address _feeReceiver, string memory _baseUri) ERC721("Veggia", "VEGGIA") Ownable(msg.sender) {
     baseURI = _baseUri;
     feeReceiver = _feeReceiver;
 }
```

The `initialize` function below correctly sets the storage in proxy as required for the protocol.

```
function initialize(address _owner, address _feeReceiver, address _capsSigner, string memory _baseUri) external {
   //...

   baseURI = _baseUri;
   //...
   feeReceiver = _feeReceiver;

   //...
}
```

Recommendation(s): update the constructor as below:

```
constructor() {
     _disableInitializers();
 }
```

Status: Acknowledged

Update from the client: Commit 658829b

6.11 [Best Practices] Unused custom error in `VeggiaERC721` contract

File(s): `VeggiaERC721.sol`

Description: The `VeggiaERC721` contract defines a custom error that is never used.

```
error ALREADY_INITIALIZED();
```

This unused error definition increases contract size and gas costs unnecessarily while providing no functional benefit to the contract.

Recommendation(s): Remove the unused error from the contract.

Status: Acknowledged

Update from the client:

7 Additional Observations

7.1 Inheriting from non-upgradeable contracts

When using a proxy pattern for upgradeability, the recommendation is to use the upgradeable version of OpenZeppelin contracts. These contracts are designed to minimize the potential for storage conflict by isolating the storage spaces for contracts in the hierarchy. The intent is to provision for upgradeability without conflicts.

It was observed that the VeggiaERC721 contract derives from a mix of upgradeable and non-upgradeable library contracts of OpenZeppelin and, hence has the potential for such risk. zKVeggies team is made aware of the risks.

8 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about zkVeggies documentation

The `zkVeggies` team was actively present in regular calls, effectively addressing concerns and questions raised by the Nethermind Security team. Additionally, the code includes NatSpec documentation for different functions and their parameters. However, the project documentation could be improved by providing a more comprehensive written overview of the system and an explanation of the different design choices.

9 Test Suite Evaluation

9.1 Compilation Output

```
$ forge build
WARNING: You are currently using Node.js v19.9.0, which is not supported by Hardhat. This can lead to unexpected
 ↪  behavior. See https://hardhat.org/nodejs-versions


Generating typings for: 80 artifacts in dir: typechain for target: ethers-v6
[] Compiling...
[] Compiling 71 files with Solc 0.8.24
[] Solc 0.8.24 finished in 3.84s
Compiler run successful!
```

9.2 Tests Output

```
$ forge test
Ran 2 tests for test/VeggiaERC721.royalties.t.sol:VeggiaERC721ERC2981Test
[PASS] test_initialRoyaltiesValues() (gas: 13704)
[PASS] test_supportsInterface() (gas: 8813)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.39ms (158.34µs CPU time)

Ran 3 tests for test/VeggiaERC721.mint3.t.sol:VeggiaERC721Mint3Test
[PASS] test_mint3() (gas: 255916)
[PASS] test_mint3Premium() (gas: 255910)
[PASS] test_mint3WhenBalanceIsUero() (gas: 31905)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 6.74ms (1.16ms CPU time)

Ran 2 tests for test/VeggiaERC721.burn.t.sol:VeggiaERC721BurnTest
[PASS] test_batchBurn() (gas: 565093)
[PASS] test_burn() (gas: 217111)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 10.89ms (8.86ms CPU time)

Ran 3 tests for test/VeggiaERC721Proxy.initialize.t.sol:VeggiaERC721ProxyInitializeTest
[PASS] test_initialize() (gas: 3683200)
[PASS] test_initializeTwice() (gas: 3686776)
[PASS] test_proxyCantReceiveEth() (gas: 3437623)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 19.80ms (8.28ms CPU time)

Ran 7 tests for test/VeggiaERC721.capsBalanceOf.t.sol:VeggiaERC721CapsBalanceOfTest
[PASS] test_capsBalanceOfShouldBeAdditionOfAllPaidBalance() (gas: 138708)
[PASS] test_capsBalanceOfShouldBeFreeMintBalance() (gas: 31546)
[PASS] test_capsBalanceOfShouldBePaidBalance() (gas: 100677)
[PASS] test_capsBalanceOfShouldBePaidBalancePlusFreeMintBalance() (gas: 105818)
[PASS] test_capsBalanceOfShouldBePaidPremiumBalance() (gas: 100721)
[PASS] test_capsBalanceOfShouldBePaidPremiumBalancePlusFreeMintBalance() (gas: 105709)
[PASS] test_capsBalanceOfShouldBeZero() (gas: 26481)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 22.54ms (2.15ms CPU time)

Ran 10 tests for test/VeggiaERC721.onlyOwner.t.sol:VeggiaERC721OnlyOwnerFctTest
[PASS] test_setBaseURI() (gas: 248523)
[PASS] test_setCapsPrice() (gas: 40955)
[PASS] test_setCapsSigner() (gas: 23809)
[PASS] test_setDefaultRoyalty() (gas: 28105)
[PASS] test_setFeeReceiver() (gas: 23264)
[PASS] test_setFreeMintCooldown() (gas: 22758)
[PASS] test_setFreeMintLimit(uint256) (runs: 256, : 26940, ~: 26985)
[PASS] test_setFreeMintLimitNotModulo3(uint256) (runs: 256, : 18836, ~: 18836)
[PASS] test_setPremiumCapsPrice() (gas: 40955)
[PASS] test_setPremiumPackPrice() (gas: 22854)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 22.48ms (33.10ms CPU time)
```

```
Ran 12 tests for test/VeggiaERC721.buyCaps.t.sol:VeggiaERC721BuyCapsTest
[PASS] test_buyCapsWrongPrice() (gas: 168573)
[PASS] test_buyNineCaps() (gas: 101423)
[PASS] test_buyNinePremiumCaps() (gas: 101448)
[PASS] test_buyPremiumCapsWrongPrice() (gas: 168445)
[PASS] test_buyTenPremiumCaps() (gas: 101404)
[PASS] test_buyThirtyCaps() (gas: 101406)
[PASS] test_buyThreeCaps() (gas: 101489)
[PASS] test_buyThreePremiumCaps() (gas: 101469)
[PASS] test_buyUnexpectedAmountOfCapsModule3(uint256) (runs: 256, : 14999, ~: 14999)
[PASS] test_buyUnexpectedAmountOfCapsNotModulo3(uint256) (runs: 256, : 14511, ~: 14511)
[PASS] test_buyUnexpectedAmountOfPremiumCapsModulo3(uint256) (runs: 256, : 14892, ~: 14892)
[PASS] test_buyUnexpectedAmountOfPremiumCapsNotModulo3(uint256) (runs: 256, : 14434, ~: 14434)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 25.65ms (68.49ms CPU time)

Ran 2 tests for test/VeggiaERC721.buyPack.t.sol:VeggiaERC721buyPremiumPackTest
[PASS] test_buyPremiumPack() (gas: 198335)
[PASS] test_buyPremiumPackWrongValue(uint256) (runs: 256, : 48833, ~: 48912)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 30.64ms (26.70ms CPU time)

Ran 5 tests for test/VeggiaERC721.open3CapsForSender.t.sol:VeggiaERC721Open3CapsForSenderTest
[PASS] test_freeMint3() (gas: 229304)
[PASS] test_freeMint3TokenIdIncrease() (gas: 215019)
[PASS] test_mint3() (gas: 263442)
[PASS] test_mint3Premium() (gas: 263505)
[PASS] test_mint3WithSignature(string,uint256,bool,address)

Ran 5 tests for test/VeggiaERC721.mint3WithSignature.t.sol:VeggiaERC721MintWithSignatureTest
[PASS] test_MintWithSignatureInvalidSender(string,uint256,bool,address) (runs: 256, : 2649286, ~: 2649212)
[PASS] test_MintWithSignatureReusedSignature(string,uint256,bool,address) (runs: 256, : 2847571, ~: 2847509)
[PASS] test_MintWithSignatureWrongSigner(string,uint256,bool,address) (runs: 256, : 2639807, ~: 2639732)
[PASS] test_mint3WithSignature(string,uint256,bool,address) (runs: 256, : 2844444, ~: 2844370)
[PASS] test_raw_mint3WithSignature() (gas: 2831186)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 174.00ms (578.92ms CPU time)

Ran 6 tests for test/VeggiaERC721.transferLock.t.sol:VeggiaERC721TransferLockTest
[PASS] test_batchBurnLockedTokens() (gas: 191112)
[PASS] test_burnLockedTokens() (gas: 191175)
[PASS] test_fuzz_tokenTransferLock(uint256) (runs: 256, : 2075722, ~: 2089633)
[PASS] test_fuzz_tokenTransferLockPaidMint(uint256) (runs: 256, : 2200887, ~: 2177326)
[PASS] test_transferFirst3PaidTokens() (gas: 247118)
[PASS] test_transferFirst3Tokens() (gas: 261223)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 249.54ms (488.27ms CPU time)

Ran 5 tests for test/VeggiaERC721.freeMint3.t.sol:VeggiaERC721FreeMintTest
[PASS] test_freeMint3WhenMoreThanTwoMintAreAvailable(uint256) (runs: 256, : 4282697, ~: 4282786)
[PASS] test_freeMint3WhenNoMintIsAvailable(uint256) (runs: 256, : 34755, ~: 34979)
[PASS] test_freeMint3WhenOneMintIsAvailable(uint256) (runs: 256, : 334784, ~: 334798)
[PASS] test_freeMint3WhenTwoMintAreAvailable() (gas: 317722)
[PASS] test_invariant_freeMint3(uint256) (runs: 256, : 316723, ~: 320658)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 2.19s (2.29s CPU time)

Ran 12 test suites in 2.20s (2.88s CPU time): 62 tests passed, 0 failed, 0 skipped (62 total tests)

Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 118.63ms (114.59ms CPU time)
```

9.2.1 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

9.2.2 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.