ARMUR Audit Report

Prepared for Wavect

PRESENTED TO

Kevin Reidl

PRESENTED BY

Akhil Sharma
Amritansh

Mar 13, 2023

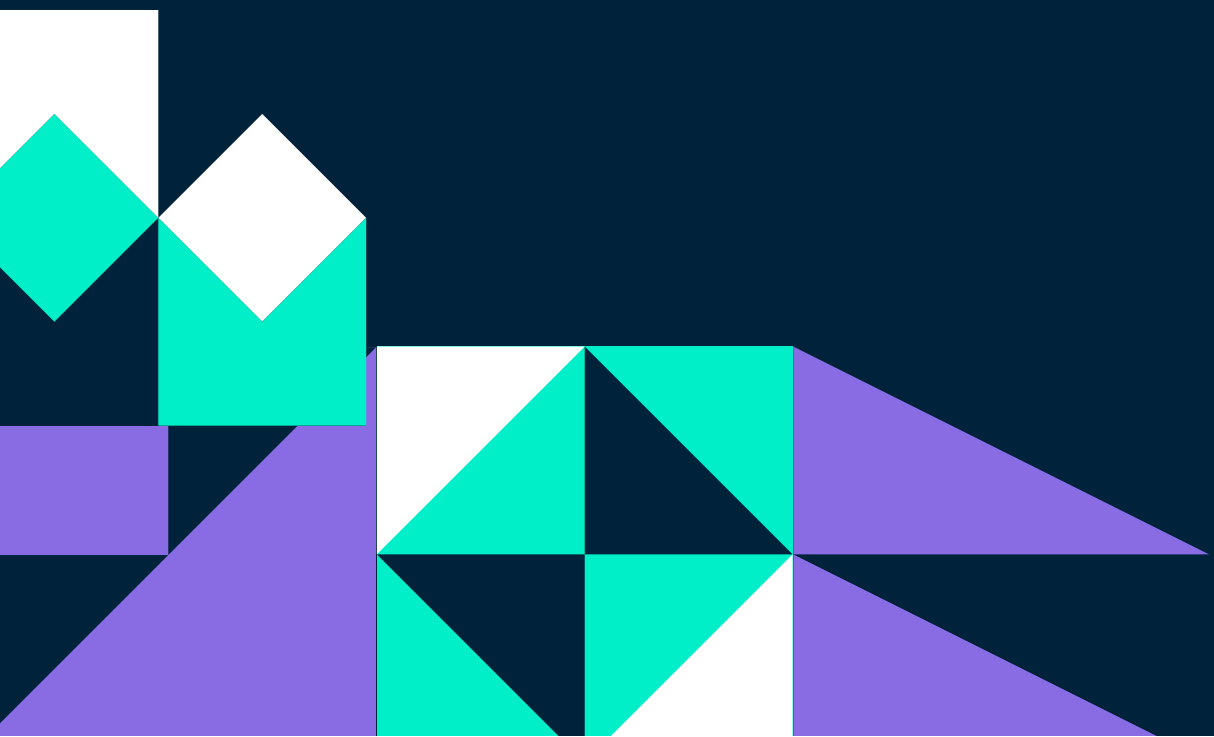
Contents

WAVECT.SOL

DOCUMENTATION

Contracts imported from OpenZeppelin:

- ERC721.sol: provides a standard interface for non-fungible tokens
- Strings.sol: provides helper functions for dealing with strings
- Pausable.sol: provides a mechanism to pause the contract in case of emergency
- SignatureChecker.sol: provides a utility function to validate signatures
- ReentrancyGuard.sol: protects against reentrant attacks
- Multicall.sol: allows multiple calls to the contract in a single transaction
- Ownable.sol: provides basic authorization control



WAVECT.SOL

DOCUMENTATION

Custom contracts:

LayerZero.sol: a contract that is not defined in this code, but is imported and used in the constructor.
LinearlyAssigned.sol: a contract that handles the sequential assignment of token IDs.

Contract variables:

publicSaleEnabled: a boolean flag that enables or disables public sale of NFTs.

licenseFrozen: a boolean flag that determines whether the license can be changed on-chain.

metadataFrozen: a boolean flag that determines whether metadata can be changed on-chain.

maxWallet: an integer that sets the maximum number of NFTs that can be minted by a single wallet.

maxTier: an integer that sets the maximum tier for yearly paid memberships.

WAVECT.SOL

DOCUMENTATION

Contract variables:

mintPrice: the price of minting a single NFT.

licenseURI: a string that contains the URI for the license associated with the NFTs.

contractURI: a string that contains the URI for the contract.

baseURI: a string that contains the URI for the base metadata of the NFTs.

subscriptionTier: a mapping that associates token IDs with their respective yearly paid membership tiers.

subscriptionPrice: a mapping that associates membership tiers with their respective prices.

minted: a mapping that associates wallet addresses with the number of NFTs they have minted.

WAVECT.SOL DOCUMENTATION

Contract functions:

constructor(): initializes the contract by setting default values and assigning ownership to a designated address.

isBelowMaxWallet(): a view function that checks whether a given wallet has exceeded the maximum number of NFTs they can mint.

isWhitelisted(): a view function that checks whether a given signature is valid.

mint(): a payable function that allows a user to mint a specified number of NFTs.

withdrawRevenue(): a function that allows the contract owner to withdraw the accumulated revenue from NFT sales.

tokenURI(): a function that returns the URI for a specific token.

changeSubscription(): a payable function that allows the owner of an NFT to change its subscription tier for a specified price.

isSubscriptionValid(): a view function that checks whether the subscription associated with a specific NFT is still valid.

ADDLIMITEDSUPPLY.SOL DOCUMENTATION

AddLimitedSupply

AddLimitedSupply is an abstract contract that tracks the supply of tokens and increments token IDs on each new mint.

Counters

Counters is a utility library in the @openzeppelin/contracts/utils/Counters.sol file. It provides a counter that can be incremented or decremented.

State Variables

_tokenCount: A Counters.Counter object that tracks the number of tokens minted so far.

_totalSupply: A uint256 variable that indicates the maximum number of tokens this token tracker will hold.

Constructor

constructor(uint256 totalSupply_): Initializes the contract by setting the maximum token count to totalSupply_.

ADDLIMITEDSUPPLY.SOL DOCUMENTATION

Public Functions

totalSupply(): Returns the maximum token count.

tokenCount(): Returns the current number of tokens minted so far.

availableTokenCount(): Returns the number of tokens that can still be minted.

nextToken(): Increments the token count and returns the next token ID. Throws an error if there are no more tokens left to mint.

Internal Functions

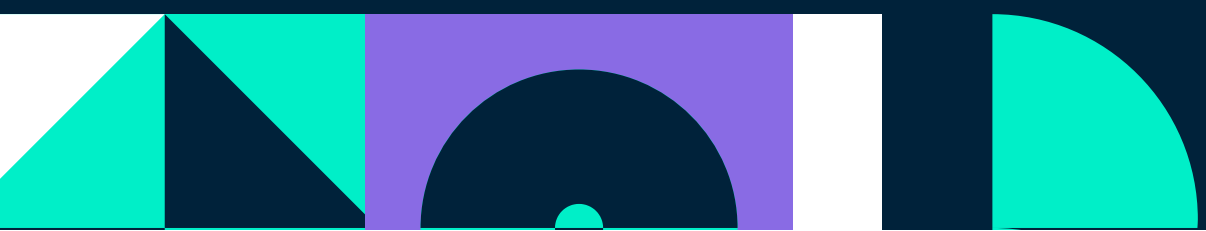
nextToken(): Increments the token count and returns the next token ID. Throws an error if there are no more tokens left to mint.

Modifiers and Events

None.

Usage

Inherit from this contract and implement the desired token functionality. Use the nextToken() function to get the next token ID and increment the token count.



LAYERZERO.SOL DOCUMENTATION

The LayerZero smart contract is an implementation of the IONFT721Core and ONFT721Core interfaces, which define the basic functionality required to interact with a non-fungible token (NFT) contract. The LayerZero contract is also a subclass of the Multicall contract from the OpenZeppelin library, which allows multiple functions to be called in a single transaction.

Contract Architecture

The LayerZero contract is designed to work in conjunction with another NFT contract, which must implement the debitFrom and creditTo functions. These functions are called by the LayerZero contract when tokens are transferred, in order to update the balances of the token holders.

The LayerZero contract is constructed with a minimum gas amount, which is used to estimate the gas cost of a transaction. The contract also stores a reference to the address of the NFT contract that it is paired with.

Contract Functions

setNFT
function setNFT(address nft_) external onlyOwner
This function sets the address of the NFT contract that this contract is paired with. It can only be called by the contract owner.

- nft_: The address of the NFT contract to pair with.

_debitFrom
function _debitFrom(address _from, uint16, bytes memory, uint256 _tokenId) internal virtual override returns (bytes memory)

This function is called by the **ONFT721Core** contract when a token is transferred from the **_from** address. It calls the **debitFrom** function on the paired NFT contract to update the token balances.

- **_from**: The address of the token holder who is transferring the token.
- **_tokenId**: The ID of the token being transferred.

_creditTo

function _creditTo(uint16, address toAddress_, uint256 tokenId_, bytes memory tier_) internal virtual override

This function is called by the ONFT721Core contract when a token is transferred to the toAddress_ address. It calls the creditTo function on the paired NFT contract to update the token balances.

- **toAddress_**: The address of the token holder who is receiving the token.
- **tokenId_**: The ID of the token being transferred.
- **tier_:** The tier data associated with the token.

Inherited Functions

The LayerZero contract inherits several functions from the IONFT721Core, ONFT721Core, and Multicall contracts. These functions provide the basic functionality required to interact with an NFT contract, as well as the ability to call multiple functions in a single transaction.

WAVECT.SOL TESTCASES

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("Wavect Contract", function () {
  let wavect;
  let owner;
  let addr1;
  let addr2;
  let addrs;

  before(async function () {
    [owner, addr1, addr2, ...addrs] = await ethers.getSigners();

    const Wavect = await ethers.getContractFactory("Wavect");
    wavect = await Wavect.deploy(
      "https://wavect.io/nft/",
      "https://wavect.io/nft/licenses/",
      "Wavect",
      "WVT",
      100,
      owner.address,
      owner.address
    );
    await wavect.deployed();
  });
```


1

```
  it("Should set the correct base URI", async function () {
    expect(await wavect.baseURI()).to.equal("https://wavect.io/nft/");
  });
```

2

```
  it("Should set the correct license URI", async function () {
    expect(await wavect.licenseURI()).to.equal(
      "https://wavect.io/nft/licenses/"
    );
  });
```

3
```
it("Should set the correct contract URI", async function () {
  expect(await wavect.contractURI()).to.equal("");
});
```

4
```
it("Should return the correct token URI", async function () {
  const tokenURI = await wavect.tokenURI(1);
  expect(tokenURI).to.equal("https://wavect.io/nft/1.json");
});
```

5
```
it("Should mint tokens", async function () {
  await wavect.mint(addr1.address, "0x", 2, {
    value: ethers.utils.parseEther("2"),
  });
```

6
```
  const balance = await wavect.balanceOf(addr1.address);
  expect(balance).to.equal(2);
});
```

7
```
it("Should not allow minting more than the max wallet", async function () {
  await expect(
    wavect.mint(addr2.address, "0x", 2, {
      value: ethers.utils.parseEther("3"),
    })
  ).to.be.revertedWith("Max wallet");
});
```


8
```
it("Should not allow minting with an invalid signature", async function () {
  const invalidSig =

"0x5d5c634cfb56fc72a3fc520a1f463045e5da4f1c6474e4a9ebc48d91396f30bb0c7aa4b244d2
f35a68a672f1e2ca9eaa98c5d5f0203b3ce3aafaa170bc41cde81b";
  await expect(
    wavect.mint(addr2.address, invalidSig, 1, {
      value: ethers.utils.parseEther("1"),
    })
  ).to.be.revertedWith("Invalid sig");
});
```

9
```
it("Should allow the owner to change the mint price", async function () {
  await expect(wavect.connect(owner).setMintPrice(ethers.utils.parseEther("1")))
    .to.emit(wavect, "SetMintPrice")
    .withArgs(ethers.utils.parseEther("1"));

  expect(await wavect.mintPrice()).to.equal(ethers.utils.parseEther("1"));
});
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.16;
import "truffle/Assert.sol";
import "../contracts/AddLimitedSupply.sol";

contract TestAddLimitedSupply {
    uint256 private constant _totalSupply = 10;

    AddLimitedSupply private _addLimitedSupply;

    function beforeEach() public {
        _addLimitedSupply = new AddLimitedSupplyMock(_totalSupply);
    }

    function testTotalSupply() public {
        Assert.equal(_addLimitedSupply.totalSupply(), _totalSupply, "Total supply should be
set to 10");
    }

    function testTokenCount() public {
        Assert.equal(_addLimitedSupply.tokenCount(), 0, "Token count should initially be set
to 0");
    }

    function testAvailableTokenCount() public {
        Assert.equal(_addLimitedSupply.availableTokenCount(), _totalSupply, "All tokens
should initially be available");
    }

    function testNextToken() public {
        uint256 expectedTokenCount = 1;
        uint256 expectedTokenId = 0;

        Assert.equal(_addLimitedSupply.nextToken(), expectedTokenId, "Token ID should be
incremented on each call");
        Assert.equal(_addLimitedSupply.tokenCount(), expectedTokenCount, "Token count
should be incremented on each call");

        // Mint all available tokens
        for (uint256 i = 1; i <= _totalSupply; i++) {
            _addLimitedSupply.nextToken();
        }

        // No more tokens should be available
        Assert.equal(_addLimitedSupply.availableTokenCount(), 0, "All tokens should have
been minted");
        Assert.reverts(_addLimitedSupply.nextToken(), "No tokens left");
    }
}

// A mock contract to expose the internal function for testing
contract AddLimitedSupplyMock is AddLimitedSupply {
    constructor (uint256 totalSupply_) AddLimitedSupply(totalSupply_) {}

    function nextToken() internal override returns (uint256) {
        return super.nextToken();
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "truffle/Assert.sol";
import "../contracts/l0/IONFT721Core.sol";
import "../contracts/LayerZero.sol";

contract TestLayerZero {

    LayerZero layerZero;

    function beforeEach() public {
        layerZero = new LayerZero(1000, address(this));
    }

    function test_setNFT() public {
        address nft = address(0x123);
        layerZero.setNFT(nft);
        Assert.equal(layerZero.nft(), nft, "NFT address was not set correctly");
    }

    function test_debitFrom() public {
        address from = address(0x456);
        uint256 tokenId = 1;

        // Mock NFT contract
        MockNFT721 nft = new MockNFT721();
        nft.mint(from, tokenId);
        layerZero.setNFT(address(nft));

        bytes memory tier = layerZero.debitFrom(from, 0, "", tokenId);

        Assert.equal(nft.ownerOf(tokenId), address(layerZero), "Token was not transferred to
LayerZero");
        Assert.equal(layerZero.tokenOwner(tokenId), from, "Token owner was not set
correctly");
        Assert.equal(tier, "", "Tier was not returned correctly");
    }
```

```
function test_creditTo() public {
        address to = address(0x789);
        uint256 tokenId = 1;
        bytes memory tier = "Tier1";

        // Mock NFT contract
6       MockNFT721 nft = new MockNFT721();
        nft.mint(address(layerZero), tokenId);
        layerZero.setNFT(address(nft));

        layerZero.creditTo(0, to, tokenId, tier);

        Assert.equal(nft.ownerOf(tokenId), to, "Token was not transferred to the recipient");
7       Assert.equal(layerZero.tokenOwner(tokenId), address(0), "Token owner was not reset");
    }
}

// Mock NFT721 contract
8 contract MockNFT721 {
    mapping(uint256 => address) private _tokenOwners;

    function mint(address to, uint256 tokenId) public {
        _tokenOwners[tokenId] = to;
    }

9   function ownerOf(uint256 tokenId) public view returns (address) {
        return _tokenOwners[tokenId];
    }

    function debitFrom(address sender, address from, uint256 tokenId) external {
10      require(ownerOf(tokenId) == from, "Token owner mismatch");
        _tokenOwners[tokenId] = sender;
    }

    function creditTo(address recipient, uint256 tokenId, bytes memory tier) external {
11      require(ownerOf(tokenId) == address(this), "Token owner mismatch");
        _tokenOwners[tokenId] = recipient;
    }
}
```

- The contract is using the SPDX license identifier "UNLICENSED", which means that the license is unspecified, making it potentially risky to use or fork.

- The contract imports multiple contracts from the OpenZeppelin library, which may introduce vulnerabilities in the code if the library is not up-to-date or has known vulnerabilities.

- The contract is using the Pausable contract from OpenZeppelin, which may cause issues if the owner or anyone else can arbitrarily pause the contract. If the contract can be paused, it may be vulnerable to a Denial of Service (DoS) attack.

- The contract is using the ReentrancyGuard contract from OpenZeppelin, which suggests that the contract may have functions that are vulnerable to reentrancy attacks.

- The contract has a variable publicSaleEnabled, which is a boolean that allows public sales. If this variable is set to true, anyone can purchase a token without being whitelisted, which may make the contract vulnerable to Sybil attacks.

- The contract has a function called mint that allows users to mint new tokens. The function does not have a nonce, which makes it possible for an attacker to replay a transaction and mint more tokens than they should be able to. This could potentially cause inflation or create other issues.

- The contract has a function called isWhitelisted that is used to check if a user is allowed to purchase tokens. The function uses ECDSA signatures to authenticate the user, but it does not validate the signature in a secure way. Specifically, the function does not check that the signature is valid for the given message hash, which may allow an attacker to bypass the whitelisting restriction.

- The contract has a mapping called minted that tracks the number of tokens that have been minted by a particular user. This mapping is not updated when tokens are burned, which may allow an attacker to bypass the max wallet restriction by minting tokens and then burning them.

- The contract has a function called withdrawRevenue that allows the contract owner to withdraw funds. This function does not have any access control mechanisms, which may allow anyone to withdraw funds if they can gain control of the contract owner's account.

- The contract has a function called tokenURI that returns the URI for a given token. This function does not check that the token actually exists, which may allow an attacker to exploit the function by passing in an arbitrary token ID.

- The contract has a function called changeSubscription that allows users to change the subscription tier for a particular token. This function does not check that the token owner has paid the required subscription fee, which may allow an attacker to exploit the function by changing the subscription tier without paying the required fee.

- The contract has a variable called maxTier that limits the number of subscription tiers. This variable is currently set to 3, but it is not clear if this value is sufficient for the contract's needs or if it may cause issues in the future.

- The contract has a variable called maxWallet that limits the number of tokens that can be minted by a particular user. This variable is currently set to 3, but it is not clear if this value is sufficient for the contract's needs or if it may cause issues in the future.

- The contract has a variable called mintPrice that sets the price for minting new tokens. This variable is currently set to a fixed value, which may make it difficult to adjust the price in response to changing market conditions.

- The contract has a variable called subscriptionPrice that sets the price for subscribing to a particular tier. This variable is currently set to a fixed value, which may make it difficult to adjust the price in response to changing market conditions.

- **Lack of input validation:** The smart contract does not perform sufficient input validation in some of its functions. For example, in the changeSubscription() function, the desired tier is not checked to ensure it is not greater than or equal to maxTier. Additionally, in the changeLicenseURI() and changeContractURI() functions, the new URIs are not validated to ensure they are not empty strings.

- **Lack of error messages:** The smart contract uses custom error types to signal certain errors, but it does not provide error messages to help users diagnose and understand the issues they are encountering. For example, the **Frozen()** and **InvalidTier()** errors do not include any information about why they were thrown.

- **Lack of event emission:** The smart contract does not emit events in some of its functions, which could make it more difficult for users to track and understand the state changes that occur. For example, the **withdrawRevenue()** function does not emit an event to signal that revenue was withdrawn.

- **Lack of access control:** The smart contract does not have sufficient access control in some of its functions. For example, the **changeMaxWallet()** and **changeMaxTier()** functions can be called by anyone, which could lead to unauthorized changes to the contract's state. Additionally, the **mint()** function can be called by anyone, even though it should only be callable by the contract owner or other authorized users.

- Use of **block.timestamp**: The **block.timestamp** variable is used to determine the expiration date of subscriptions. However, this can be manipulated by miners to some extent, which could lead to unexpected behavior. It would be more secure to use a more reliable source of time, such as an external oracle.

- Use of **tx.origin**: The **tx.origin** variable is used to check whether a user is whitelisted in the **isWhitelisted()** function. However, this can be spoofed by attackers, which could lead to unauthorized access to the contract. It would be more secure to use **msg.sender** instead.

ADDLIMITEDSUPPLY.SOL ISSUES

- **Access Control:** The contract does not include any access control mechanisms, which means that anyone can call the contract's functions. This could lead to malicious actors manipulating the contract's state or stealing its funds.

- **Integer Overflow and Underflow**: The contract uses unsigned integers to represent the total supply and token count. If the contract allows minting beyond the maximum value of uint256, it could lead to an integer overflow or underflow, causing unexpected behavior and potentially resulting in lost funds.

- **Lack of Event Emitting**: The contract does not emit events, which makes it difficult to track contract interactions and could hinder its usability for third-party applications that rely on events.

- **Code Upgrades:** The contract is not upgradeable, which means that any bugs or security issues cannot be fixed without creating a new contract and migrating the data. This can be costly and time-consuming for the contract owner and may lead to disruption for users.

- **Lack of Function Modifiers**: The contract does not use function modifiers to restrict the access to certain functions. This may lead to security vulnerabilities if the contract owner does not properly restrict access to certain functions.

ADDLIMITEDSUPPLY.SOL ISSUES

- **Lack of Input Validation**: The contract does not validate input parameters, which may lead to unexpected behavior and vulnerabilities if users provide malicious input.

- **Gas Limit:** The contract does not include any gas limit protections, which means that a user could potentially launch a denial-of-service (DoS) attack by sending a transaction with a large amount of gas.

- **Dependency Risks:** The contract relies on an external library imported from OpenZeppelin. If this library has any vulnerabilities, it could lead to security issues with this contract.

- **Logic Flaws**: The contract may have logic flaws, which could allow attackers to exploit unintended behavior or edge cases. For example, if the token count is not properly updated, it may result in the creation of duplicate tokens.

- **Lack of validation of input parameters:** The smart contract does not validate the input parameters passed to the constructor, such as totalSupply_, to ensure that they are valid values. This could lead to unexpected behavior or errors if invalid input values are used.

- **No restrictions on who can mint tokens:** The smart contract does not include any logic to restrict who can mint tokens. This means that anyone who has access to the contract's mint() function can mint tokens, which could be problematic if unauthorized users are able to mint tokens.

ADDLIMITEDSUPPLY.SOL ISSUES

- **Inability to burn tokens:** The smart contract does not include any logic to allow tokens to be burned (i.e., permanently destroyed). This means that once a token has been minted, it cannot be removed from circulation. This could be problematic if there are issues with the token that need to be corrected or if a user wants to permanently destroy a token for some reason.

- **Lack of event logging**: The smart contract does not include any events to log important actions or state changes. This could make it difficult to track the history of the contract's state and actions, which could be problematic for auditing and debugging purposes.

- **No validation of token ownership**: The smart contract does not include any logic to validate that a user who is attempting to transfer or interact with a token actually owns that token. This could allow users to perform actions on tokens that they do not own, leading to potential security vulnerabilities or other issues.

- **No expiration date for tokens**: The smart contract does not include any logic to expire or remove tokens after a certain amount of time. This could be problematic if the tokens are only intended to be valid for a limited period of time, as they could remain in circulation indefinitely.

LAYERZERO.SOL ISSUES

- **Inheritance:** The contract inherits from two contracts - IONFT721Core and ONFT721Core. Any vulnerabilities in those contracts may also impact the security of the LayerZero contract.

- **Access control:** The setNFT function can only be called by the contract owner, which is implemented through the onlyOwner modifier. However, it is unclear who the owner is and how they are set. Additionally, there are other functions that don't have any access control implemented, making them open to potential attacks.

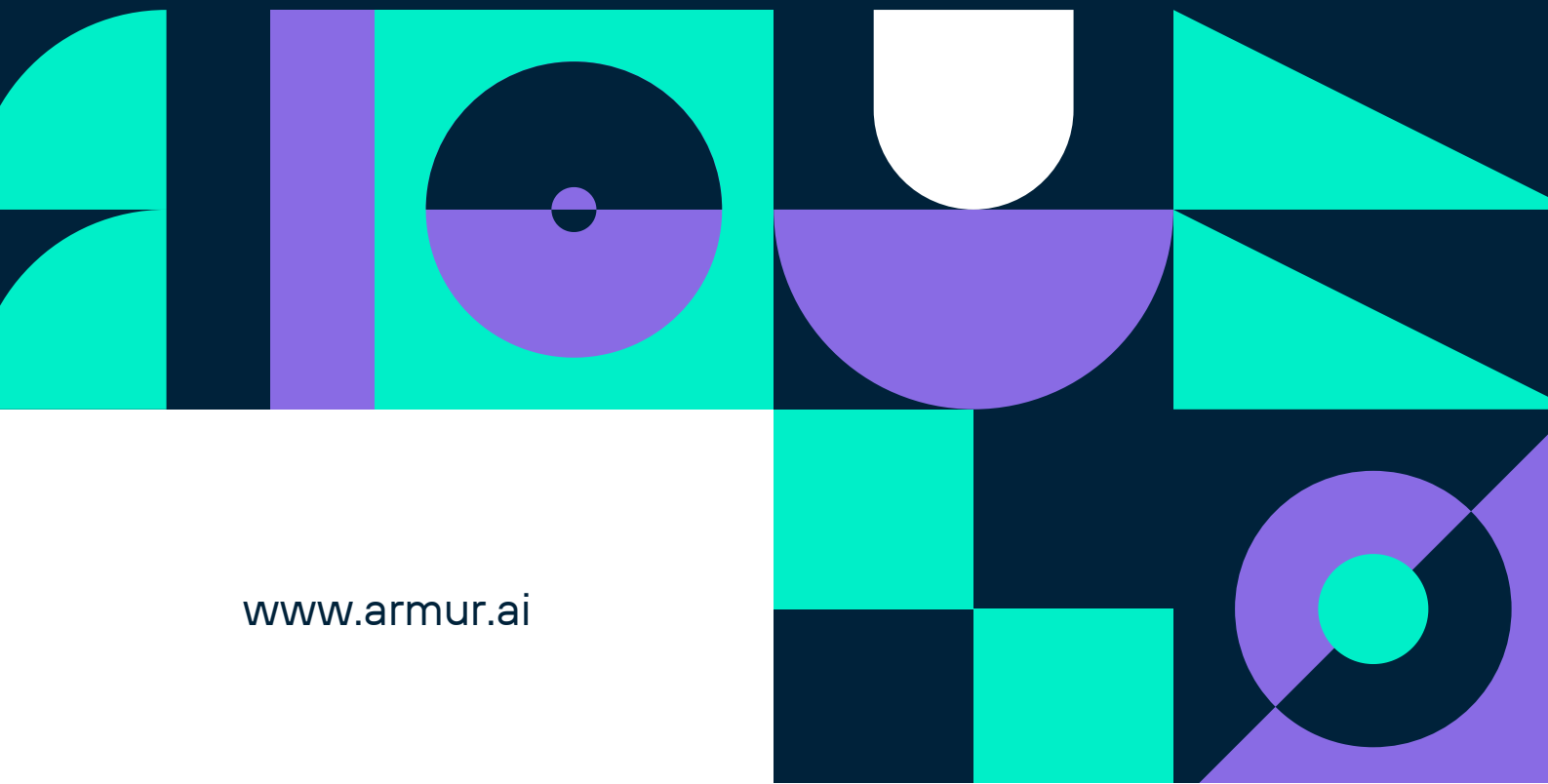
- **External contract interaction:** The debitFrom and creditTo functions interact with an external contract (nft). The nft address is set through the setNFT function, which means that if an attacker gains control of the setNFT function, they could potentially direct the debitFrom and creditTo functions to interact with a malicious contract. The nft contract's implementation is not provided in the code, so it is not possible to determine if the interaction with the contract is secure.

- **Error handling:** The debitFrom and creditTo functions use a require statement to check if the interaction with the nft contract was successful. However, if the nft contract reverts with a custom error message, it is lost, and the error message is not propagated to the user.

- **Function overriding:** The LayerZero contract overrides the _debitFrom and _creditTo functions from the IONFT721Core and ONFT721Core contracts, respectively. If these functions are not overridden correctly or if the parent contracts change in a way that is not compatible with the overriding functions, it could result in unexpected behavior and potential vulnerabilities.

- **Multicall:** The LayerZero contract imports the Multicall contract from the OpenZeppelin library. While Multicall itself is a widely used and trusted contract, there is always the possibility that vulnerabilities could be introduced in future updates to the library.

Thanks.



www.armur.ai