Escher contest
Findings & Analysis Report

2023-02-22

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Escher smart contract system written in Solidity. The audit contest took place between December 6—December 9 2022.

Wardens

124 Wardens contributed reports to the Escher contest:

1. 0x1f8b
2. 0x446576
3. 0x4non
4. 0x52
5. 0xA5DF
6. 0xDave
7. 0xDecorativePineapple
8. 0xNazgul
9. 0xRobocop
10. 0xSmartContract
11. 0xbepresent

12. 0xdeadbeef0x
13. 0xfuje
14. 0xhacksmithh
15. 8olidity
16. AkshaySrivastav
17. Awesome
18. Aymen0909
19. Bnke0x0
20. CRYP70
21. Ch_301
22. Chom
23. Diana
24. Dinesh11G
25. Englave
26. ForkEth (NullbutC00L and filipaze)
27. Franfran
28. HollaDieWaldfee
29. JC
30. Josiah
31. KingNFT
32. Lambda
33. MHKK33
34. Madalad
35. Matin
36. Parth
37. Rahoz
38. RaymondFam
39. ReyAdmirado
40. Ruhum

41. Soosh
42. TomJ
43. Tricko
44. Tutturu
45. UniversalCrypto (amaechieth and tettehnetworks)
46. _Adam
47. __141345__
48. adriro
49. ahmedov
50. ajtra
51. asgeir
52. aviggiano
53. bin2chen
54. btk
55. c3phas
56. carrotsmuggler
57. cccz
58. chaduke
59. cryptonue
60. cryptostellar5
61. csanuragjain
62. danyams
63. dic0de
64. evan
65. ey88
66. eyexploit
67. fatherOfBlocks
68. fs0c
69. gasperpre

70. gz627
71. gzeon
72. hansfriese
73. helios
74. hihen
75. imare
76. immeas
77. jadezti
78. jayphbee
79. joestakey
80. jonatascm
81. kaliberpoziomka8552
82. karanctf
83. kiki_dev
84. kree-dotcom
85. ladboy233
86. lukris02
87. lumoswiz
88. mahdikarimi
89. martin
90. minhquanym
91. minhtrng
92. nalus
93. nameruse
94. neumo
95. nicobevi
96. obront
97. oyc_109
98. pashov

99. pauliax
100. pfapostol
101. poirots (DavideSilva, resende, naps62 and eighty)
102. reassor
103. rvierdiiev
104. saian
105. sakshamguruji
106. seyni
107. shark
108. simon135
109. slvDev
110. sorrynotsorry
111. stealthyz
112. supernova
113. tnevler
114. tourist
115. wait
116. yellowBirdy
117. yixxas
118. zapaz
119. zaskoh

This contest was judged by berndartmueller.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 16 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 13 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 34 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 17 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Escher contest repository, and is composed of 12 smart contracts written in the Solidity programming language and includes 534 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (3)

[H-01] selfdestruct may cause the funds to be lost

Submitted by bin2chen, *also found by* pauliax

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/FixedPrice.sol#L110

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/OpenEdition.sol#L122

Impact

After the contract is destroyed, the subsequent execution of the contract's `#buy()` will always succede, the msg.value will be locked in this address.

Proof of Concept

When FixedPrice.sol and OpenEdition.sol are finished, `selfdestruct()` will be executed to destroy the contract.

But there is a problem with this:

Suppose when Alice and Bob execute the purchase transaction at the same time, the transaction is in the memory pool (or Alice executes the transaction, but Bob is still operating the purchase in the UI, the UI does not know that the contract has been destroyed)

If Alice meets the finalId, the contract is destroyed after her transaction ends.

Note: "When there is no code at the address, the transaction will succeed, and the msg.value will be stored in the contract. Although no code is executed."

After that, Bob's transaction will be executed.

This way the msg.value passed by Bob is lost and locked forever in the address of this empty code.

Suggestion: don't use selfdestruct, use modified state to represent that the contract has completed the sale.

Recommended Mitigation Steps

```
contract FixedPrice is Initializable, OwnableUpgradeable, ISale
...
    function _end(Sale memory _sale) internal {
        emit End(_sale);
        ISaleFactory(factory).feeReceiver().transfer(address(thi
-       selfdestruct(_sale.saleReceiver);
+       sale.finalId = sale.currentId
+       sale.saleReceiver.transfer(address(this).balance);

    }
```

berndartmueller (judge) commented:

> The warden demonstrates an issue that leads to loss of user funds in case the contract is destroyed with `selfdestruct`.

> This clearly differentiates from the other submissions outlining the deprecated use of `selfdestruct` due to EIP-4758.

stevennevins (Escher) disputed and commented:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "forge-std/Test.sol";
import {EscherTest, FixedPrice, FixedPriceFactory} from "test/Fi

contract Issue296Test is EscherTest {
    FixedPrice.Sale public fixedSale;
    FixedPriceFactory public fixedSales;

    FixedPrice public sale;

    function setUp() public virtual override {
        super.setUp();
        fixedSales = new FixedPriceFactory();
        fixedSale = FixedPrice.Sale({
            currentId: uint48(0),
            finalId: uint48(10),
            edition: address(edition),
            price: uint96(uint256(1 ether)),
```

```
            saleReceiver: payable(address(69)),
            startTime: uint96(block.timestamp)
        });
        sale = FixedPrice(fixedSales.createFixedSale(fixedSale))
        edition.grantRole(edition.MINTER_ROLE(), address(sale));
        sale.buy{value: 10 ether}(10);
    }

    function test_RevertsWhenAfterSelfDestruct_Buy() public {
        vm.expectRevert();
        sale.buy{value: 1 ether}(1);
    }
}
```

> Any purchases after the selfdestruct would revert as shown above. In order to accurately mock this in Foundry, you can't perform the destruction within a test function since everything would be one large transaction and the code would still exist as far as Foundry is concerned because it's all in the same call context. So you have to move the purchase that selfdestructs the contract into the setup function to demonstrate the behavior.

berndartmueller (judge) commented:

> @stevennevins You're right with your Foundry example. However, there are subtle nuances to this that make this finding valid.

> In your Foundry test, the Solidity function call via `sale.buy{value: 1 ether}(1)` at the end, uses the `extcodesize` opcode to check that the contract that is about to be called actually exists and causes an exception if it does not.

> However, in the above demonstrated PoC, if a transaction gets executed *after* the contract got destroyed (due to buying the final token id), the transaction does not revert and any value sent along with the transaction will be stored in the contract.

> I was able to reproduce this behavior with a minimal example contract on Goerli via Remix:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.7;

contract Sale  {
```

```
        function buy() external payable {
            _end();
        }

        function _end() internal {
            selfdestruct(payable(msg.sender));
        }
    }
```

> Calling the `Sale.buy()` function in Remix twice, once with a value of `1337 wei` and once with `333 wei`:

1. https://goerli.etherscan.io/tx/0x138d2c061ed547ec5bfd4d7d9b03b519f92b5f63a74b73c45a1af6efe202727c
2. https://goerli.etherscan.io/tx/0x7ca9032cf79f0a8e29da17061e7eeafb46edfddff96633dc9bbf5957f1aff142

> As you can see, the first transaction will destroy the contract, `1337 wei` is sent to the beneficiary (caller). The second transaction succeeds as well. However, the value of `333 wei` is kept at the address of the destroyed contract.

> After many considerations, I consider **High** to be the appropriate severity. Multiple parallel transactions (e.g. front-runners) will likely try to buy the last NFT IDs, whereas all transactions executed after the last successful buy will render the sent `msg.value` locked in the destroyed contract. Those users lost funds while not receiving any NFTs.

[H-02] `LPDA` price can underflow the price due to bad settings and potentially brick the contract

Submitted by adriro, *also found by* Aymen0909, Franfran, kiki_dev, Ch_301, Chom, slvDev, lukris02, 0xRobocop, minhquanym, immeas, nameruse, 0xDecorativePineapple, carrotsmuggler, kaliberpoziomka8552, jonatascm, minhtrng, imare, neumo, ladboy233, Tricko, mahdikarimi, sorrynotsorry, kree-dotcom, pauliax, poirots, bin2chen, jayphbee, 0xDave, jadezti, evan, reassor, gz627, 0xbepresent, 0xA5DF, hihen, chaduke, hansfriese, yixxas, Madalad, HollaDieWaldfee, Parth, 0x446576, lumoswiz, danyams, obront, zapaz, rvierdiiev, 8olidity, *and* Ruhum

The dutch auction in the `LPDA` contract is implemented by configuring a start price and price drop per second.

A bad set of settings can cause an issue where the elapsed duration of the sale multiplied by the drop per second gets bigger than the start price and underflows the current price calculation.

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/LPDA.sol#L117

```
function getPrice() public view returns (uint256) {
    Sale memory temp = sale;
    (uint256 start, uint256 end) = (temp.startTime, temp.endTime
    if (block.timestamp < start) return type(uint256).max;
    if (temp.currentId == temp.finalId) return temp.finalPrice;

    uint256 timeElapsed = end > block.timestamp ? block.timestam
    return temp.startPrice - (temp.dropPerSecond * timeElapsed);
}
```

This means that if `temp.dropPerSecond * timeElapsed > temp.startPrice` then the unsigned integer result will become negative and underflow, leading to potentially bricking the contract and an eventual loss of funds.

Impact

Due to Solidity 0.8 default checked math, the subtraction of the start price and the drop will cause a negative value that will generate an underflow in the unsigned integer type and lead to a transaction revert.

Calls to `getPrice` will revert, and since this function is used in the `buy` to calculate the current NFT price it will also cause the buy process to fail. The price drop will continue to increase as time passes, making it impossible to recover from this situation and effectively bricking the contract.

This will eventually lead to a loss of funds because currently the only way to end a sale and transfer funds to the sale and fee receiver is to buy the complete set of

NFTs in the sale (i.e. buy everything up to the `sale.finalId`) which will be impossible if the `buy` function is bricked.

Proof of Concept

In the following test, the start price is 1500 and the duration is 1 hour (3600 seconds) with a drop of 1 per second. At about ~40% of the elapsed time the price drop will start underflowing the price, reverting the calls to both `getPrice` and `buy`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "forge-std/Test.sol";
import {FixedPriceFactory} from "src/minters/FixedPriceFactory.s
import {FixedPrice} from "src/minters/FixedPrice.sol";
import {OpenEditionFactory} from "src/minters/OpenEditionFactory
import {OpenEdition} from "src/minters/OpenEdition.sol";
import {LPDAFactory} from "src/minters/LPDAFactory.sol";
import {LPDA} from "src/minters/LPDA.sol";
import {Escher721} from "src/Escher721.sol";

contract AuditTest is Test {
    address deployer;
    address creator;
    address buyer;

    FixedPriceFactory fixedPriceFactory;
    OpenEditionFactory openEditionFactory;
    LPDAFactory lpdaFactory;

    function setUp() public {
        deployer = makeAddr("deployer");
        creator = makeAddr("creator");
        buyer = makeAddr("buyer");

        vm.deal(buyer, 1e18);

        vm.startPrank(deployer);

        fixedPriceFactory = new FixedPriceFactory();
        openEditionFactory = new OpenEditionFactory();
        lpdaFactory = new LPDAFactory();
```

```
        vm.stopPrank();
    }

    function test_LPDA_getPrice_NegativePrice() public {
        // Setup NFT and create sale
        vm.startPrank(creator);

        Escher721 nft = new Escher721();
        nft.initialize(creator, address(0), "Test NFT", "TNFT");

        // Duration is 1 hour (3600 seconds), with a start price
        uint48 startId = 0;
        uint48 finalId = 1;
        uint80 startPrice = 1500;
        uint80 dropPerSecond = 1;
        uint96 startTime = uint96(block.timestamp);
        uint96 endTime = uint96(block.timestamp + 1 hours);

        LPDA.Sale memory sale = LPDA.Sale(
            startId, // uint48 currentId;
            finalId, // uint48 finalId;
            address(nft), // address edition;
            startPrice, // uint80 startPrice;
            0, // uint80 finalPrice;
            dropPerSecond, // uint80 dropPerSecond;
            endTime, // uint96 endTime;
            payable(creator), // address payable saleReceiver;
            startTime // uint96 startTime;
        );
        LPDA lpdaSale = LPDA(lpdaFactory.createLPDASale(sale));

        nft.grantRole(nft.MINTER_ROLE(), address(lpdaSale));

        vm.stopPrank();

        // simulate we are in the middle of the sale duration
        vm.warp(startTime + 0.5 hours);

        vm.startPrank(buyer);

        // getPrice will revert due to the overflow caused by th
        vm.expectRevert();
        lpdaSale.getPrice();

        // This will also cause the contract to be bricked, sinc
```

```
            uint256 amount = 1;
            uint256 price = 1234;
            vm.expectRevert();
            lpdaSale.buy{value: price * amount}(amount);

            vm.stopPrank();
        }
    }
```

Recommendation

Add a validation in the `LPDAFactory.createLPDASale` function to ensure that the given duration and drop per second settings can't underflow the price.

```
    require((sale.endTime - sale.startTime) * sale.dropPerSecond <=
```

stevennevins (Escher) confirmed

[H-03] `saleReceiver` and `feeReceiver` can steal refunds after sale has ended

Submitted by immeas, *also found by* AkshaySrivastav, cryptonue, saian, minhquanym, 0xDecorativePineapple, gzeon, ey88, wait, pauliax, HollaDieWaldfee, jayphbee, bin2chen, evan, reassor, fs0c, hihen, hansfriese, 0x52, *and* aviggiano

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/LPDA.sol#L67-L68

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/LPDA.sol#L81-L88

First, lets go over how a buy happens.

A buyer can buy NFTs at a higher price and then once the auction ends they can use `refund()` to return the over payments. The effect is that they bought the NFTs at the lowest price (Lowest Price Dutch Auction).

Now, let's move on to what happens when the sale ends:

The sale is considered ended when the last NFT is sold which triggers the payout to the seller and fee collector:

```
81:         if (newId == temp.finalId) {
82:             sale.finalPrice = uint80(price);
83:             uint256 totalSale = price * amountSold;
84:             uint256 fee = totalSale / 20;
85:             ISaleFactory(factory).feeReceiver().transfer(fee)
86:             temp.saleReceiver.transfer(totalSale - fee);
87:             _end();
88:         }
```

Earlier there's also a check that you cannot continue buying once the `currentId` has reached `finalId`:

```
67:         uint48 newId = amount + temp.currentId;
68:         require(newId <= temp.finalId, "TOO MANY");
```

However, it is still possible to buy `0` NFTs for whichever price you want even after the sale has ended. Triggering the "end of sale" snippet again, since `newId` will still equal `temp.finalId`.

The attacker, `saleReceiver` (or `feeReceiver`), buys `0` NFTs for the delta between `totalSale` and the balance still in the contract (the over payments by buyers). If there is more balance in the contract than `totalSales` this can be iterated until the contract is empty.

The attacker has then stolen the over payments from the buyers.

A buyer can mitigate this by continuously calling `refund()` as the price lowers but that would incur a high gas cost.

Impact

`saleReceiver` or `feeReceiver` can steal buyers over payments after the sale has ended. Who gains the most depends on circumstances in the auction.

Proof of Concept

PoC test in `test/LPDA.t.sol`:

```
    function test_BuyStealRefund() public {
        sale = LPDA(lpdaSales.createLPDASale(lpdaSale));
        edition.grantRole(edition.MINTER_ROLE(), address(sale));

        // buy most nfts at a higher price
        sale.buy{value: 9 ether}(9);

        // warp to when price is lowest
        vm.warp(block.timestamp + 1 days);
        uint256 price = sale.getPrice(); // 0.9 eth

        // buy last nft at lowest possible price
        sale.buy{value: price}(1);

        uint256 contractBalanceAfterEnd = address(sale).balance;
        uint256 receiverBalanceAfterEnd = address(69).balance;
        console.log("Sale end");
        console.log("LPDA contract",contractBalanceAfterEnd); //
        console.log("saleReceiver ",receiverBalanceAfterEnd); //

        // buy 0 nfts for the totalSales price - current balance
        // totalSales: 9 eth - contract balance 0.9 eth = ~8.1 e
        uint256 totalSale = price * 10;
        uint256 delta = totalSale - contractBalanceAfterEnd;
        sale.buy{value: delta}(0);

        console.log("after steal");
        console.log("LPDA contract",address(sale).balance);
        console.log("saleReceiver ",address(69).balance - receiv

        // buyer supposed to get back the ~0.9 eth
        vm.expectRevert(); // EvmError: OutOfFund
        sale.refund(); // nothing to refund
    }
```

Tools Used

VS Code, Forge

Recommended Mitigation Steps

I can think of different options of how to mitigate this:

- Don't allow buying 0 NFTs
- Don't allow buying if `newId == finalId` since the sale has ended

mehtaculous (Escher) disagreed with severity

Medium Risk Findings (13)

[M-01] Escher721 contract does not have setTokenRoyalty function

Submitted by **HollaDieWaldfee**, *also found by* **0xNazgul**, **hansfriese**, **cccz**, *and* **obront**

On the Code4rena page of this contest there is a "SALES PATTERNS" section that describes the flow of how to use Sales:

https://code4rena.com/contests/2022-12-escher-contest

It contains this statement:

> If the artist would like sales and royalties to go somewhere other than the default royalty receiver, they
> must call setTokenRoyalty with the following variables

So an artist should be able to call the `setTokenRoyalty` function on the `Escher721` contract.

However this function cannot be called. It does not exist. There exists a `_setTokenRoyalty` in `ERC2981.sol` from OpenZeppelin. This function however is `internal` (https://github.com/OpenZeppelin/openzeppelin-

contracts/blob/3d7a93876a2e5e1d7fe29b5a0e96e222afdc4cfa/contracts/token/common/ERC2981.sol#L94).

So there is no `setTokenRoyalty` function that can be called by the artist. So an artist cannot set a royalty individually for a token as is stated in the documentation.

Proof of Concept

I tried calling `setTokenRoyalty` from inside `Escher721.t.sol` with the following test:

```
function test_setDefaultRoyalty() public {
    edition.setTokenRoyalty(1,address(this), 5);
}
```

This won't even compile because the `setTokenRoyalty` function does not exist.

Tools Used

VS Code

Recommended Mitigation Steps

In order to expose the internal `_setTokenRoyalty` function to the artist, add the following function to the `Escher721` contract:

```
function setTokenRoyalty(
    uint256 tokenId,
    address receiver,
    uint96 feeNumerato r
) public onlyRole(DEFAULT_ADMIN_ROLE) {
    _setTokenRoyalty(tokenId,receiver,feeNumerator);
}
```

mehtaculous (Escher) disputed

[M-02] Use of `payable.transfer()` Might Render ETH Impossible to Withdraw

Submitted by RaymondFam, *also found by* ajtra, AkshaySrivastav, JC, asgeir, pashov, cryptonue, carrotsmuggler, fs0c, karanctf, jonatascm, Tutturu, ladboy233, Awesome, pauliax, Rahoz, bin2chen, shark, supernova, __141345__, simon135, lumoswiz, zaskoh, 0xdeadbeef0x, ajtra, hansfriese, yellowBirdy, cccz, CRYP70, dic0de, Bnke0x0, tourist, ahmedov, Parth, ahmedov, obront, chaduke, zapaz, HollaDieWaldfee, btk, rvierdiiev, martin, 0xhacksmithh, fatherOfBlocks, *and* aviggiano

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/LPDA.sol#L105

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/LPDA.sol#L85-L86

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/FixedPrice.sol#L109

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/OpenEdition.sol#L92

Impact

The protocol uses Solidity's `transfer()` when transferring ETH to the recipients. This has some notable shortcomings when the recipient is a smart contract, which can render ETH impossible to transfer. Specifically, the transfer will inevitably fail when the smart contract:

- does not implement a payable fallback function, or
- implements a payable fallback function which would incur more than 2300 gas units, or
- implements a payable fallback function incurring less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

Proof of Concept

File: LPDA.sol

```
85:             ISaleFactory(factory).feeReceiver().transfer(fee)
86:             temp.saleReceiver.transfer(totalSale - fee);
```

```
105:         payable(msg.sender).transfer(owed);
```

[File: FixedPrice.sol#L109]

```
109:         ISaleFactory(factory).feeReceiver().transfer(address
```

[File: OpenEdition.sol#L92]

```
92:         ISaleFactory(factory).feeReceiver().transfer(address
```

Issues pertaining to the use of `transfer()` in the code blocks above may be referenced further via:

- CONSENSYS Diligence's article
- OpenZeppelin news & events

Recommended Mitigation Steps

Using `call` with its returned boolean checked in combination with re-entrancy guard is highly recommended after December 2019.

For instance, line 105 in `LPDA.sol` may be refactored as follows:

```
- payable(msg.sender).transfer(owed);
+ (bool success, ) = payable(msg.sender).call{ value: owed }('')
+ require(success, " Transfer of ETH Failed");
```

Alternatively, `Address.sendValue()` available in OpenZeppelin Contract's Address library can be used to transfer the Ether without being limited to 2300 gas units.

And again, in either of the above measures adopted, the risks of re-entrancy stemming from the use of this function can be mitigated by tightly following the

"Check-effects-interactions" pattern and/or using OpenZeppelin Contract's ReentrancyGuard contract.

mehtaculous (Escher) confirmed and commented:

> Agree with severity. Solution would be to attempt to transfer ETH, and if that is unsuccessful, transfer WETH instead.

[M-03] `OpenEdition.buy()` might revert due to uint overflow when it should work.

Submitted by hansfriese, also found by AkshaySrivastav

`OpenEdition.buy()` might revert due to uint overflow when it should work.

Proof of Concept

`OpenEdition.buy()` validates the total funds like below.

```
function buy(uint256 _amount) external payable {
    uint24 amount = uint24(_amount);
    Sale memory temp = sale;
    IEscher721 nft = IEscher721(temp.edition);
    require(block.timestamp >= temp.startTime, "TOO SOON");
    require(block.timestamp < temp.endTime, "TOO LATE");
    require(amount * sale.price == msg.value, "WRONG PRICE")
```

Here, `amount` was declared as `uint24` and `sale.price` is `uint72`.

And it will revert when `amount * sale.price >= type(uint72).max` and such cases would be likely to happen e.g. `amount = 64(so 2^6), sale.price = 73 * 10^18(so 2^66)`.

As a result, `buy()` might revert when it should work properly.

Recommended Mitigation Steps

We should modify like below.

```
        require(uint256(amount) * sale.price == msg.value, "WRONG PR
```

stevennevins (Escher) confirmed

[M-04] Editions should be checked if they are actually deployed from the legitimate Escher721Factory

Submitted by hansfriese, *also found by* carrotsmuggler, imare, nalus, 0xRobocop, Englave, Dinesh11G, hihen, danyams, MHKK33, Ruhum, *and* ahmedov

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/FixedPriceFactory.sol#L29

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/LPDAFactory.sol#L29

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/OpenEditionFactory.sol#L29

Impact

For all kinds of sales, the creators create new sales contracts with arbitrary sale data, and the edition is not properly checked.

Malicious creators can create fake contracts that implemented `IEscher721` and fake buyers to get free earnings.

Proof of Concept

Sales contracts can be created by any creator and the sale data is filled with the one provided by the creator.

The protocol does not validate the `sale.edition` provided by the creator and malicious creators can effectively use their fake contract address that implemented `IEscher721`.

In the worst case, buyers will not get anything after their payments.

```
29
30 function createFixedSale(FixedPrice.Sale calldata _sale) exte
31   require(IEscher721(_sale.edition).hasRole(bytes32(0x00), ms
32   require(_sale.startTime >= block.timestamp, "START TIME IN
33   require(_sale.finalId > _sale.currentId, "FINAL ID BEFORE C
34
35   clone = implementation.clone();
36   FixedPrice(clone).initialize(_sale);
37
38   emit NewFixedPriceContract(msg.sender, _sale.edition, clone
39 }
```

Malicious creators can use a fake contract as an edition to steal funds from users.

Recommended Mitigation Steps

Track all the deployed `Escher721` contracts in the `Escher721Factory.sol` and validate the `sale.edition` before creating sales contracts.

stevennevins (Escher) disagreed with severity and commented:

> Marking as Medium, because funds are not directly at risk but this is a good point for us to keep in mind.

berndartmueller (judge) decreased severity to Medium

[M-05] Inconsistency in fees

Submitted by csanuragjain

If the seller cancels before the sale startTime then all funds should be moved to saleReceiver without any deduction (Assuming seller has sent some ETH accidentally

before sell start). But in `FixedPrice.sol` contract, fees are deducted even when seller cancels before the sale startTime which could lead to loss of funds.

Proof of Concept

1. A new sale is started
2. Seller selfdestructed one of his personal contracts and by mistake gave this sale contract as receiver. This forcefully sends the remaining 20 ETH to the `FixedPrice.sol` contract.
3. Seller realizes his mistake and tries to cancel the sale as sale as not yet started using the `cancel` function.

```
function cancel() external onlyOwner {
        require(block.timestamp < sale.startTime, "TOO LATE");
        _end(sale);
    }
```

4. This internally calls the `\_end` function

```
function _end(Sale memory _sale) internal {
        emit End(_sale);
        ISaleFactory(factory).feeReceiver().transfer(address(thi
        selfdestruct(_sale.saleReceiver);
    }
```

5. The `_end` function deducts fees of 20/20=1 ETH even though seller has cancelled before the sale starts.

Recommended Mitigation Steps

Revise the `cancel` function

```
function cancel() external onlyOwner {
        require(block.timestamp < sale.startTime, "TOO LATE");
        emit End(_sale);
        selfdestruct(_sale.saleReceiver);
    }
```

stevennevins (Escher) disagreed with severity and commented:

> Marking as Low as it seems pretty unlikely of a scenario.

berndartmueller (judge) commented:

> I consider Medium severity appropriate because fees are sent to the receiver even though a sale has not started yet. This also clearly deviates from the implementation in the `FixedPrice` contract, where fees are not sent in case the owner cancels a sale.

[M-06] `buy()` in `LPDA.sol` Can be Manipulated by Buyers

Submitted by RaymondFam, *also found by* Josiah

A buyer could plan on buying early at higher prices to make sure he would secure a portion (say 50%) of NFTs he desired. When the number of NFTs still available got smaller and that `sale.endTime` were yet to hit, he would then watch the mempool and repeatedly attempt to thwart the final group of buyers from successfully completing their respective transactions amidst the efforts to prolong the Dutch auction till `sale.endTime` was reached.

Proof of Concept

Assuming this particular edition pertained to a 100 NFT collection that would at most last for 60 minutes, and Bob planned on minting 10 of them. At the beginning of the Dutch auction, he would first mint 5 NFTs at higher prices no doubt. At 50th minute, `sale.currentId == 95`. Alice, upon seeing this, made up her mind and proceeded to buy the remaining NFTs. Bob, seeing this transaction queuing in the mempool, invoked `buy()` to mint 1 NFT by sending in higher amount of gas to front run Alice. Needless to say, Alice's transaction was going to revert on line 68 because `newId == 101`:

File: LPDA.sol#L68

```
68:         require(newId <= temp.finalId, "TOO MANY");
```

Noticing the number of NFTs still available had become 4, Alice attempted to mint the remaining 4 NFTs this time. Bob, upon seeing the similar queue in the mempool again, front ran Alice with another mint of 1 NFT.

These steps were repeatedly carried out until Bob managed to get all NFTs he wanted where the last one was minted right after `sale.endTime` hit. At this point, every successful buyer was happy to get the biggest refund possible ensuring that each NFT was only paid for the lowest price. This intended goal, on the contrary, was achieved at the expense of the seller getting the lowest amount of revenue and that the front run buyers minting nothing.

Recommended Mitigation Steps

Consider refactoring the affected code line as follows:

```
- require(newId <= temp.finalId, "TOO MANY");
+ if(newId > temp.finalId) {
+       uint256 diff = newId - temp.finalId;
+       newId = temp.finalId;
+       amountSold -= diff;
+       amount -= diff;
+ }
```

stevennevins (Escher) confirmed

[M-07] ETH will get stuck if all NFTs do not get sold.

Submitted by AkshaySrivastav, *also found by* seyni, tnevler, wait, nalus, immeas, gzeon, lukris02, minhtrng, kiki_dev, adriro, Soosh, KingNFT, mahdikarimi, ladboy233, jonatascm, kree-dotcom, csanuragjain, _Adam, hihen, jadezti, reassor, gz627, 0xA5DF, 0xdeadbeef0x, nameruse, danyams, hansfriese, Madalad, gasperpre, yixxas, Parth, lumoswiz, obront, eyexploit, rvierdiiev, *and* 0x52

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/FixedPrice.sol#L73

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/LPDA.sol#L81-L88

Impact

In the `buy` function of FixedPrice and LPDA contracts, the transfer of funds to `saleReceiver` only happens when `newId` becomes equal to `finalId`, i.e, when the entire batch of NFTs gets sold completely.

FixedPrice:

```
if (newId == sale_.finalId) _end(sale);
```

LPDA:

```
if (newId == temp.finalId) {
    sale.finalPrice = uint80(price);
    uint256 totalSale = price * amountSold;
    uint256 fee = totalSale / 20;
    ISaleFactory(factory).feeReceiver().transfer(fee);
    temp.saleReceiver.transfer(totalSale - fee);
    _end();
}
```

This logic can cause issues if only a subset of NFTs get sold. In that case the ETH collected by contract from the sales of NFTs will get stuck inside the contract.

There is no function inside those contracts to recover the ETH collected from the partial sale of the NFT batch. There is also no function to send back those ETH to the NFT buyers. This causes the ETH to be in a stuck state as they cannot be pulled from the contracts.

The only way to recover those ETH is to explicitly buy the entire batch of to-be-sold NFTs so that the contract automatically triggers the sale ending conditions. This recovery method may require a decent amount of upfront capital to buy all unsold NFTs.

Proof of Concept

Consider this scenario:

- The sale contract deployer deploys a sale contract (FixedSale) to sell 100 NFTs ranging from id 1 to 100 with price 1 ETH per NFT.
- Users only buy the initial 10 NFTs. So the sale contract now has 10 ETH and currently 90 NFTs are unsold.
- No more users buy anymore NFTs from the sale contract.
- The 10 ETH are in stuck state as the `saleReceiver` cannot pull them out even though some NFTs have been already legitimately sold.
- To recover those 10 ETH, the contract deployer or `saleReceiver` has to come up with 90 ETH and buy the unsold 90 NFTs.

The scenario for the LPDA sale contract is similar. ETH can get stuck for partial batch NFT sales and upfront capital will be needed to recover the ETH.

Recommended Mitigation Steps

The contracts can have an additional function to claim the ETH collected from already successful NFT sales.

```
function claim() external onlyOwner {
    ISaleFactory(factory).feeReceiver().transfer(address(thi
    sale.saleReceiver.transfer(address(this).balance);
}
```

Please note, use of `transfer` is not recommended. The above code is just an example mitigation.

mehtaculous (Escher) disagreed with severity and commented:

> The creator could simply buy the remaining IDs and then receive their ETH in the same transaction, effectively unsticking the sale. Not considered High priority, but do agree there should be a better cancellation pattern for when all the NFTs do not get sold.

berndartmueller (judge) decreased severity to Medium and commented:

> The demonstrated finding has an external requirement (too many token ids, low buy pressure,...), and there exists the possibility for the creator to buy the

> remaining, unsold IDs (which may or may not be expensive). Therefore, I am downgrading the finding to Medium severity.

[M-08] Unsafe downcasting operation truncate user's input

Submitted by ladboy233, *also found by* seyni, slvDev, minhquanym, gzeon, karanctf, Matin, 0x1f8b, yixxas, UniversalCrypto, obront, *and* rvierdiiev

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/LPDA.sol#L71

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/LPDA.sol#L82

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/LPDA.sol#L101

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/FixedPrice.sol#L62

Impact

Unsafe downcasting operation truncates user's input.

Proof of Concept

There are a few unsafe downcasting operation that truncates user's input. The impact can be severe or minimal.

In FixedPrice.sol,

```
    /// @notice buy from a fixed price sale after the sale starts
    /// @param _amount the amount of editions to buy
    function buy(uint256 _amount) external payable {
            Sale memory sale_ = sale;
            IEscher721 nft = IEscher721(sale_.edition);
```

```
        require(block.timestamp >= sale_.startTime, "TOO SOON");
        require(_amount * sale_.price == msg.value, "WRONG PRICE
        uint48 newId = uint48(_amount) + sale_.currentId;
        require(newId <= sale_.finalId, "TOO MANY");

        for (uint48 x = sale_.currentId + 1; x <= newId; x++) {
                nft.mint(msg.sender, x);
        }

        sale.currentId = newId;

        emit Buy(msg.sender, _amount, msg.value, sale);

        if (newId == sale_.finalId) _end(sale);
}
```

The amount is unsafely downcasted from uint256 to uint48, note the code:

```
require(_amount * sale_.price == msg.value, "WRONG PRICE");
uint48 newId = uint48(_amount) + sale_.currentId;
```

The upper limit for uint48 is 281474976710655,

If user wants to buy more than 281474976710655 amount of NFT and pay the 281474976710655 * sale price amount, the user can only receive 281474976710655 amount of NFT because of the downcasting.

In LPDA.sol, we are unsafely downcasting the price in the buy function:

```
receipts[msg.sender].amount += amount;
receipts[msg.sender].balance += uint80(msg.value);

for (uint256 x = temp.currentId + 1; x <= newId; x++) {
        nft.mint(msg.sender, x);
}

sale.currentId = newId;

emit Buy(msg.sender, amount, msg.value, temp);

if (newId == temp.finalId) {
```

```
        sale.finalPrice = uint80(price);
        uint256 totalSale = price * amountSold;
        uint256 fee = totalSale / 20;
        ISaleFactory(factory).feeReceiver().transfer(fee);
        temp.saleReceiver.transfer(totalSale - fee);
        _end();
    }
```

Note the line:

uint80(msg.value) and uint80(price)

In LPDA.sol, same issue exists in the refund function:

```
    /// @notice allow a buyer to get a refund on the current price c
    function refund() public {
        Receipt memory r = receipts[msg.sender];
        uint80 price = uint80(getPrice()) * r.amount;
        uint80 owed = r.balance - price;
        require(owed > 0, "NOTHING TO REFUND");
        receipts[msg.sender].balance = price;
        payable(msg.sender).transfer(owed);
    }
```

Note the downcasting: `uint80(getPrice())`

This means if the price goes above uint80, it will be wrongly truncated to uint80 for price.

The Downcasting in LPDA.sol is damaging because it truncated user's fund.

Below is the POC:

Add test in LPDA.t.sol

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/test/LPDA.t.sol#L167

```
    function test_LPDA_downcasting_POC() public {

        // make the lpda sales contract
        sale = LPDA(lpdaSales.createLPDASale(lpdaSale));
        // authorize the lpda sale to mint tokens
        edition.grantRole(edition.MINTER_ROLE(), address(sale));
        //lets buy an NFT

        uint256 val = uint256(type(uint80).max) + 10 ether;
        console.log('msg.value');
        console.log(val);
        sale.buy{value: val}(1);

    }
```

And import “forge-std/console.sol” in LPDA.sol:

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/LPDA.sol#L3

```
import "forge-std/console.sol";
```

And add console.log in LPDA.sol buy function.

https://github.com/code-423n4/2022-12-escher/blob/5d8be6aa0e8634fdb2f328b99076b0d05fefab73/src/minters/LPDA.sol#L69

```
    receipts[msg.sender].amount += amount;
    receipts[msg.sender].balance += uint80(msg.value);

    console.log("truncated value");
    console.log(receipts[msg.sender].balance);
```

We run our test:

```
forge test -vv --match test_LPDA_downcasting_POC
```

The output is:

```
Running 1 test for test/LPDA.t.sol:LPDATest
[PASS] test_LPDA_downcasting_POC() (gas: 385619)
Logs:
  msg.value
  120893581961462917470617 5
  truncated value
  999999999999999999

Test result: ok. 1 passed; 0 failed; finished in 3.61ms
```

As we can see, user uses 120893581961462917470617 5 to buy in LPDA.sol but the balance is truncated to 999999999999999999, later user is not able to get the refund they are entitled to because the msg.value is unsafely downcasted.

Also note the downcasting for getPrice in LPDA.sol is also a issue:

The getPrice in LPDA.sol returns a uint256

```
/// @notice the price of the sale
function getPrice() public view returns (uint256) {
        Sale memory temp = sale;
        (uint256 start, uint256 end) = (temp.startTime, temp.enc
        if (block.timestamp < start) return type(uint256).max;
        if (temp.currentId == temp.finalId) return temp.finalPri

        uint256 timeElapsed = end > block.timestamp ? block.time
        return temp.startPrice - (temp.dropPerSecond * timeElaps
}
```

But this is downcasted into uint80 in function buy and refund.

```
uint80 price = uint80(getPrice()) * r.amount;
```

Recommended Mitigation Steps

We recommend the project handle downcasting and use safe casting library to make sure the downcast does not provide an unexpected truncate value.

https://docs.openzeppelin.com/contracts/3.x/api/utils#SafeCast

stevennevins (Escher) confirmed, but disagreed with severity and commented:

> Scenarios seem pretty unlikely but we should just handle these truncations better.

berndartmueller (judge) commented:

> While very unlikely, it could result in a loss of funds for the user. Therefore, I consider Medium severity to be appropriate.

[M-09] `selfdestruct()` will not be available after EIP-4758

Submitted by tnevler, *also found by* yixxas, pashov, 0xRobocop, 0xbepresent, Ruhum, Chom, 0xDecorativePineapple, lukris02, Soosh, imare, *and* yellowBirdy

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/FixedPrice.sol#L110

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/OpenEdition.sol#L122

Impact

`selfdestruct()` will not be available after EIP-4758. This EIP will rename the SELFDESTRUCT opcode and replace its functionality. It will no longer destroy code or storage, so, the contract still will be available.

In this case it will break the logic of the project because it will not work as expected:

FixedPrice.sol

- After calling the cancel() function, the contract still will be available. Users will be able to buy a number of NFTs even if the current sale is cancelled.

OpenEdition.sol

- After calling the cancel() function, the contract still will be available. Users will be able to buy a number of NFTs even if the current sale is cancelled.

Proof of Concept

According to EIP-4758:

- The SELFDESTRUCT opcode is renamed to SENDALL, and now only immediately moves all ETH in the account to the target; it no longer destroys code or storage or alters the nonce.
- All refunds related to SELFDESTRUCT are removed.

Recommended Mitigation Steps

The architecture should be changed to avoid that problem.

stevennevins (Escher) disagreed with severity and commented:

> If the storage is still intact, then after this point, the start and end time would prevent buys from occurring. But yeah I agree we should move away from selfdestruct all together.

> Suggesting Low

berndartmueller (judge) commented:

> This finding demonstrates an issue regarding the `EIP-4758`, which makes it impossible to `cancel` an active sale. Thus, I consider Medium severity to be appropriate as the functionality of the protocol is impacted.

[M-10] Sale contracts can be bricked if any other minter mints a token with an id that overlaps the sale

Submitted by adriro, *also found by* Ch_301, 0xRobocop, carrotsmuggler, neumo, HollaDieWaldfee, jonatascm, AkshaySrivastav, KingNFT, bin2chen, hihen, 0xA5DF, hansfriese, gasperpre, cccz, *and* lumoswiz

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/FixedPrice.sol#L66

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/OpenEdition.sol#L67

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/LPDA.sol#L74

In all of the three sale types of contracts, the `buy` function will mint tokens with sequential ids starting at a given configurable value.

If an external entity mints any of the token ids involved in the sale, then the buy procedure will fail since it will try to mint an already existing id. This can be the creator manually minting a token or another similar contract that creates a sale.

Taking the `FixedPrice` contract as the example, if any of the token ids between `sale.currentId + 1` and `sale.finalId` is minted by an external entity, then the `buy` process will be bricked since it will try to mint an existing token id. This happens in lines 65-67:

https://github.com/code-423n4/2022-12-escher/blob/main/src/minters/FixedPrice.sol#L65-L67

```
    for (uint48 x = sale_.currentId + 1; x <= newId; x++) {
        nft.mint(msg.sender, x);
    }
```

The implementation of the `mint` function (OZ contracts) requires that the token id doesn't exist:

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/token/ERC721/ERC721Upgradeable.sol#L29

```
function _mint(address to, uint256 tokenId) internal virtual {
    require(to != address(0), "ERC721: mint to the zero address'
    require(!_exists(tokenId), "ERC721: token already minted");
    ...
```

Impact

If any of the scenarios previously described happens, then the contract will be bricked since it will try to mint an already existing token and will revert the transaction. There is no way to update the token ids in an ongoing sale, which means that the buy function will always fail for any call.

This will also cause a loss of funds in the case of the `FixedPrice` and `LPDA` contracts since those two require that all token ids are sold before funds can be withdrawn.

Proof of Concept

The following test reproduces the issue using the `FixedPrice` sale contract type.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "forge-std/Test.sol";
import {FixedPriceFactory} from "src/minters/FixedPriceFactory.s
import {FixedPrice} from "src/minters/FixedPrice.sol";
import {OpenEditionFactory} from "src/minters/OpenEditionFactory
import {OpenEdition} from "src/minters/OpenEdition.sol";
import {LPDAFactory} from "src/minters/LPDAFactory.sol";
import {LPDA} from "src/minters/LPDA.sol";
import {Escher721} from "src/Escher721.sol";

contract AuditTest is Test {
    address deployer;
    address creator;
    address buyer;

    FixedPriceFactory fixedPriceFactory;
    OpenEditionFactory openEditionFactory;
```

```
    LPDAFactory lpdaFactory;

    function setUp() public {
        deployer = makeAddr("deployer");
        creator = makeAddr("creator");
        buyer = makeAddr("buyer");

        vm.deal(buyer, 1e18);

        vm.startPrank(deployer);

        fixedPriceFactory = new FixedPriceFactory();
        openEditionFactory = new OpenEditionFactory();
        lpdaFactory = new LPDAFactory();

        vm.stopPrank();
    }

    function test_FixedPrice_buy_MintBrickedContract() public {
        // Suppose there's another protocol or account that acts
        address otherMinter = makeAddr("otherMinter");

        // Setup NFT and create sale
        vm.startPrank(creator);

        Escher721 nft = new Escher721();
        nft.initialize(creator, address(0), "Test NFT", "TNFT");

        uint48 startId = 10;
        uint48 finalId = 15;
        uint96 price = 1;

        FixedPrice.Sale memory sale = FixedPrice.Sale(
            startId, // uint48 currentId;
            finalId, // uint48 finalId;
            address(nft), // address edition;
            price, // uint96 price;
            payable(creator), // address payable saleReceiver;
            uint96(block.timestamp) // uint96 startTime;
        );
        FixedPrice fixedSale = FixedPrice(fixedPriceFactory.crea

        nft.grantRole(nft.MINTER_ROLE(), address(fixedSale));
        nft.grantRole(nft.MINTER_ROLE(), otherMinter);

        vm.stopPrank();
```

```
            // Now, other minter mints at least one of the ids inclu
            vm.prank(otherMinter);
            nft.mint(makeAddr("receiver"), startId + 2);

            // Now buyer goes to sale and tries to buy the NFTs
            vm.startPrank(buyer);

            // The following will revert. The contract will be brick
            uint256 amount = finalId - startId;
            vm.expectRevert("ERC721: token already minted");
            fixedSale.buy{value: price * amount}(amount);

            vm.stopPrank();
        }
    }
```

Recommended Mitigation Steps

The quick solution would be to mint the tokens to the sale contract, but that will require a potentially high gas usage if the sale involves a large amount of tokens.

Other alternatives would be to "reserve" a range of tokens to a particular minter (and validate that each one mints over their own range) or to have a single minter role at a given time, so there's just a single entity that can mint tokens at a given time.

mehtaculous (Escher) disputed, disagreed with severity and commented:

> This is considered more of a Medium risk and a solution would be to have the ERC-721 contract manage the sequential IDs and have the sales contracts simply request minting of the next token ID.

berndartmueller (judge) decreased severity to Medium and commented:

> Due to the external requirement of this finding, I'm downgrading it to Medium severity.

[M-11] Creator can still "cancel" a sale after it has started by revoking permissions in `OpenEdition` contract

Submitted by adriro, also found by 0xA5DF and HollaDieWaldfee

The `OpenEdition` type of sale has a start time and an end time. The creator (or owner of the contract) can cancel a sale using the `cancel` function only if it hasn't started yet (i.e. start time is after current block timestamp).

However, the NFT creator can still revoke the minting permissions to the sale contract if he wishes to pull out of the sale. This will prevent anyone from calling the `buy` and prevent any further sale from the collection.

Impact

The owner of the sale contract can still virtually cancel a sale after it has started by simply revoking the minting permissions to the sale contract.

This will cause the `buy` function to fail because the call to `mint` will revert, effectively making it impossible to further purchase NFTs and causing the effect of canceling the sale.

Proof of Concept

In the following test, the creator of the sale decides to pull from it in the middle of the elapsed duration. After that, he only needs to wait until the end time passes to call `finalize` and end the sale.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "forge-std/Test.sol";
import {FixedPriceFactory} from "src/minters/FixedPriceFactory.s
import {FixedPrice} from "src/minters/FixedPrice.sol";
import {OpenEditionFactory} from "src/minters/OpenEditionFactory
import {OpenEdition} from "src/minters/OpenEdition.sol";
import {LPDAFactory} from "src/minters/LPDAFactory.sol";
import {LPDA} from "src/minters/LPDA.sol";
import {Escher721} from "src/Escher721.sol";

contract AuditTest is Test {
    address deployer;
    address creator;
    address buyer;
```

```
FixedPriceFactory fixedPriceFactory;
OpenEditionFactory openEditionFactory;
LPDAFactory lpdaFactory;

function setUp() public {
    deployer = makeAddr("deployer");
    creator = makeAddr("creator");
    buyer = makeAddr("buyer");

    vm.deal(buyer, 1e18);

    vm.startPrank(deployer);

    fixedPriceFactory = new FixedPriceFactory();
    openEditionFactory = new OpenEditionFactory();
    lpdaFactory = new LPDAFactory();

    vm.stopPrank();
}

function test_OpenEdition_buy_CancelSaleByRevokingRole() pub
    // Setup NFT and create sale
    vm.startPrank(creator);

    Escher721 nft = new Escher721();
    nft.initialize(creator, address(0), "Test NFT", "TNFT");

    uint24 startId = 0;
    uint72 price = 1e6;
    uint96 startTime = uint96(block.timestamp);
    uint96 endTime = uint96(block.timestamp + 1 hours);

    OpenEdition.Sale memory sale = OpenEdition.Sale(
        price, // uint72 price;
        startId, // uint24 currentId;
        address(nft), // address edition;
        startTime, // uint96 startTime;
        payable(creator), // address payable saleReceiver;
        endTime // uint96 endTime;
    );
    OpenEdition openSale = OpenEdition(openEditionFactory.cr

    nft.grantRole(nft.MINTER_ROLE(), address(openSale));

    vm.stopPrank();
```

```
        // simulate we are in the middle of the sale duration
        vm.warp(startTime + 0.5 hours);

        // Now creator decides to pull out of the sale. Since he
        vm.startPrank(creator);

        nft.revokeRole(nft.MINTER_ROLE(), address(openSale));

        vm.stopPrank();

        vm.startPrank(buyer);

        // Buyer can't call buy because sale contract can't mint
        uint256 amount = 1;
        vm.expectRevert();
        openSale.buy{value: price * amount}(amount);

        vm.stopPrank();

        // Now creator just needs to wait until sale ends
        vm.warp(endTime);

        vm.prank(creator);
        openSale.finalize();
    }
}
```

Recommendation

One possibilty is to acknowledge the fact that the creator still has control over the minting process and can arbitrarily decide when to cancel the sale.

If this route is taken, then the recommendation would be to make the `cancel` function unrestricted.

Preminting the NFTs is not a solution because of high gas costs and the fact that the amount of tokens to be sold is not known beforehand, it's determined by the actual amount sold during the sale.

A more elaborate solution would require additional architecture changes to prevent the revocation of the minting permissions if some conditions (defined by target sale

contract) are met.

stevennevins (Escher) confirmed

[M-12] NFTs mintable after Auction deadline expires

Submitted by ForkEth, *also found by* Ch_301, Chom, minhquanym, adriro, csanuragjain, CRYP70, *and* Lambda

The `buy` function on the `LPDA.sol` contract is not validating if the auction is still running, allowing a purchase to be made after the stipulated time. The `endtime` variable used to store the end date of the auction is not used at any point to validate whether the purchase is being made within the deadline.

Proof of Concept

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

import "forge-std/Test.sol";
import {EscherTest} from "./utils/EscherTest.sol";
import {LPDAFactory, LPDA} from "src/minters/LPDAFactory.sol";

contract LPDABase is EscherTest {
        LPDAFactory public lpdaSales;
        LPDA.Sale public lpdaSale;

        function setUp() public virtual override {
                super.setUp();
                lpdaSales = new LPDAFactory();
                // set up a LPDA Sale
                lpdaSale = LPDA.Sale({
                        currentId: uint48(0),
                        finalId: uint48(10),
                        edition: address(edition),
                        startPrice: uint80(uint256(1 ether)),
                        finalPrice: uint80(uint256(0.1 ether)),
                        dropPerSecond: uint80(uint256(0.1 ether)
                        startTime: uint96(block.timestamp),
```

```
                        saleReceiver: payable(address(69)),
                        endTime: uint96(block.timestamp + 1 days
                    });
            }
    }

    contract LPDATest is LPDABase {
            LPDA public sale;

            event End(LPDA.Sale _saleInfo);
            function test_Buy() public {
                    sale = LPDA(lpdaSales.createLPDASale(lpdaSale));
                    // authorize the lpda sale to mint tokens
                    edition.grantRole(edition.MINTER_ROLE(), address
                    vm.warp(block.timestamp + 3 days);
                    sale.buy{value: 1 ether}(1);
                    assertEq(address(sale).balance, 1 ether);
            }
    }
```

The code above shows that even after two days after the `endTime` it was still possible to make the purchase.

Recommended Mitigation Steps

Our recommendation would be to introduce a require to validate if the purchase is being made within the `endTime` .

```
    require(block.timestamp > sale.endTime, "TOO LATE");
```

The above must be placed at the beginning of the `buy` function.

[mehtaculous (Escher) confirmed and commented](#):

> Agree that this is Medium priority. We want to check that we are not past the `endTime` on the `buy()` function.

[M-13] Ownership of EscherERC721.sol contracts can be changed, thus creator roles become useless

Submitted by stealthyz

https://github.com/code-423n4/2022-12-escher/blob/main/src/Escher.sol#L11

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/65420cb9c943c32eb7e8c9da60183a413d90067a/contracts/access/AccessControlUpgradeable.sol#L150

https://github.com/code-423n4/2022-12-escher/blob/main/src/Escher721Factory.sol#L32

Impact

(creator = has a CREATOR_ROLE in Escher.sol

non-creator = doesn't have a CREATOR_ROLE in Escher.sol)

Currently creating an ERC721 edition via the `Escher721Factory.sol` contract requires a user to have the `CREATOR_ROLE` in the main `Escher.sol` contract.

This requirement would mean that only users with the aforementioned role can be admins of editions. This requirement can be bypassed by having a 'malicious' creator create an edition for someone who doesn't have the `CREATOR_ROLE` set by creating the edition and granting the `DEFAULT_ADMIN_ROLE` to the non-creator via AccessControl.sol's `grantRole()` function. This way the non-creator can revoke the original creator's roles in this edition and gain full ownership. Now this non-creator admin can create sales and operate as if he/she was a creator.

This defeats the point of having a role for creators and makes this function of the protocol not as described == faulty.

Proof of Concept

A creator can benefit from his role by taking in payments for creating ERC721 editions for other people. This would make sense so that his risk can be covered.

1. A creator gets onboarded to Escher.
2. For some time he stays good but then people start offering payments for edition ownership
3. The more creators there are in Escher, the less of a chance to get caught. But then again, the more inclusive Escher gets, the more people will pay to get their own edition, which makes this pretty dangerous
4. This creator creates an edition with the payer's inputs and grants the payer the DEFAULT*ADMIN*ROLE
5. Payer revokes all of the creator's roles and becomes the new admin

You can edit the Escher721.t.sol file to look like this and then run the test normally, everything should go through without errors:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "forge-std/Test.sol";
import {EscherTest} from "./utils/EscherTest.sol";

contract Escher721Test is EscherTest {
    function setUp() public override {
        super.setUp();
    }
    function test_grantRoles() public {
        address _this = address(this);
        // Malicious creator grants someone else the rights for
        edition.grantRole(bytes32(0x0), address(9));
        vm.prank(address(9));
        // Now this user can grant/revoke roles
        edition.grantRole(edition.MINTER_ROLE(), address(9));
        assertEq(edition.hasRole(bytes32(0x0), address(9)), true
        // clean out the partner
        edition.revokeRole(bytes32(0x0), _this);
        assertEq(edition.hasRole(bytes32(0x0), _this), false);
    }
}
```

This kind of attack/abuse is currently hard to track. There is no centralized database of created editions and their admins at the time of creations (i.e. a mapping). This makes it hard to track down malicious creators who create editions for other people.

Looping through the emitted events and comparing current admins to the emitted admins is a hassle especially if this protocol gains a lot of traction in the future which I assume is the end goal here.

Tools Used

VS Code, Forge

Recommended Mitigation Steps

In `EscherERC721.sol` implementation contract, it is recommended to override the `grantRole()` function of `AccessControlUpgradeable.sol` with something like:

```
    function grantRole(bytes32 role, address account) internal overr
        revert("Admins can't be chagned");
    }
```

This will disable the granting of roles after initialization. The initialization function already has the required granting of roles done and they cannot be changed after this fix.

Overall it would be recommended to store the created editions in a mapping for example to prevent problems like these.

stevennevins (Escher) confirmed

Low Risk and Non-Critical Issues

For this contest, 34 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: TomJ, seyni, joestakey, ajtra, slvDev, carrotsmuggler, saian, 0xfuje, nameruse, immeas, ladboy233, adriro, 0xNazgul, 0xA5DF, _Adam, 0xRobocop, hansfriese, 0x4non, zaskoh, sakshamguruji, 0x1f8b, yixxas, HollaDieWaldfee, cccz, Lambda, helios, RaymondFam, obront, gasperpre, Bnke0x0, rvierdiiev, oyc_109, *and* danyams.

Low Risk Issues Summary

Number	Issues Details	Context
[L-01]	Use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` contract	3
[L-02]	Use `safeTransferOwnership` instead of `transferOwnership` function	1
[L-03]	Owner can renounce Ownership	3
[L-04]	Critical Address Changes Should Use Two-step Procedure	4
[L-05]	Loss of precision due to rounding	3
[L-06]	`setFeeReceiver` value check is missing in critical Set Functions	3

Total: 6 issues

[L-01] Use `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` contract

FixedPrice.sol#L10

LPDA.sol#L10

OpenEdition.sol#L10

`transferOwnership` function is used to change Ownership from `OwnableUpgradeable.sol`.

There is another Openzeppelin Ownable contract (Ownable2StepUpgradeable.sol) has `transferOwnership` function, use is more secure due to 2-stage ownership transfer.

Ownable2StepUpgradeable.sol

[L-02] Use `safeTransferOwnership` instead of `transferOwnership` function

FixedPriceFactory.sol#L9

LPDAFactory.sol#L9

OpenEditionFactory.sol#L9

`transferOwnership` function is used to change Ownership from `Ownable.sol`.

Use a 2 structure transferOwnership which is safer. `safeTransferOwnership`, use is more secure due to 2-stage ownership transfer.

Recommended Mitigation Steps

Use `Ownable2Step.sol`

Ownable2Step.sol

[L-03] Owner can renounce Ownership

FixedPriceFactory.sol#L9

LPDAFactory.sol#L9

OpenEditionFactory.sol#L9

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The Openzeppelin's Ownable used in this project contract implements renounceOwnership. This can represent a certain risk if the ownership is renounced for any other reason than by design. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

`onlyOwner` function:

```
    function setFeeReceiver(address payable fees) public onlyOwner
```

Recommended Mitigation Steps

We recommend to either reimplement the function to disable it or to clearly specify if it is part of the contract design.

[L-04] Critical Address Changes Should Use Two-step Procedure

The critical procedures should be a two-step process.

4 results, 3 files

```
src/minters/FixedPriceFactory.sol:
  48:     function setFeeReceiver(address payable fees) public c
  49          feeReceiver = fees;

src/minters/LPDAFactory.sol:
  46:     function setFeeReceiver(address payable fees) public c
  47          feeReceiver = fees;

src/minters/OpenEditionFactory.sol:
  42:     function setFeeReceiver(address payable fees) public c
  43          feeReceiver = fees;
```

Recommended Mitigation Steps

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two-step procedure on the critical functions.

[L-05] Loss of precision due to rounding

Although the loss of precision is minimal, this information can be specified in the Natspec comments and documentation.

```
3 results 3 files

src/minters/FixedPrice.sol:
  111:          ISaleFactory(factory).feeReceiver().transfer(addr

src/minters/LPDA.sol:
  85:             uint256 fee = totalSale / 20;
  86              ISaleFactory(factory).feeReceiver().transfer(f
```

```
src/minters/OpenEdition.sol:
  92:         ISaleFactory(factory).feeReceiver().transfer(addre
```

[L-06] `setFeeReceiver` value check is missing in critical Set Functions

If `onlyOwner` enter 0 address in `setFeeReceiver` (it is imposible), platform loses Money, so should add to address check require in function.

```
3 results - 3 files

src/minters/FixedPriceFactory.sol:
  48:     function setFeeReceiver(address payable fees) public c
  49:         feeReceiver = fees;
  50:     }
  51  }

src/minters/LPDAFactory.sol:
  46:     function setFeeReceiver(address payable fees) public c
  47:         feeReceiver = fees;
  48:     }
  49  }

src/minters/OpenEditionFactory.sol:
  42:     function setFeeReceiver(address payable fees) public c
  43:         feeReceiver = fees;
  44:     }
  45  }
```

Non-Critical Issues Summary

Number	Issues Details	Context
[N-01]	Insufficient coverage	
[N-02]	Test arguments and actual contract arguments are incompatible	1
[N-03]	Constant values such as a call to keccak256(), should used to immutable rather than constant	4

Number	Issues Details	Context
[N-04]	For functions, follow Solidity standard naming conventions	1
[N-05]	`Function writing` that does not comply with the `Solidity Style Guide`	All contracts
[N-06]	Lack of event emission after critical `initialize()` function	1
[N-07]	Add a timelock to critical functions	3
[N-08]	`Empty blocks` should be *removed* or *Emit* something	1
[N-09]	NatSpec comments should be increased in contracts	All contracts
[N-10]	Floating pragma	All contracts
[N-11]	Add parameter to Event-Emit	3
[N-12]	Omissions in Events	3
[N-13]	Lack of event emission after critical `initialize()` functions	1
[N-14]	NatSpec is missing	11
[N-15]	Initial value check is missing in Set Functions	5
[N-16]	Protect your NFT from copying in POW forks	

Total: 16 issues

[N-01] Insufficient coverage

The test coverage rate of the project is 72%. Testing all functions is best practice in terms of security criteria.

```
| File                                | % Lines           | % Stat
|-------------------------------------|-------------------|------
| src/Escher.sol                      | 80.00% (8/10)     | 80.00%
| src/Escher721.sol                   | 15.38% (2/13)     | 15.38%
| src/Escher721Factory.sol            | 0.00% (0/6)       | 0.00%
| src/minters/FixedPrice.sol          | 82.61% (19/23)    | 86.21%
| src/minters/FixedPriceFactory.sol   | 100.00% (7/7)     | 100.00
| src/minters/LPDA.sol                | 91.11% (41/45)    | 91.53%
```

```
| src/minters/LPDAFactory.sol         | 100.00% (11/11)  | 100.0(
| src/minters/OpenEdition.sol         | 81.48% (22/27)   | 84.85%
| src/minters/OpenEditionFactory.sol | 100.00% (7/7)    | 100.0(
| src/uris/Base.sol                   | 0.00% (0/3)      | 0.00%
| src/uris/Generative.sol             | 0.00% (0/8)      | 0.00%
| src/uris/Unique.sol                 | 0.00% (0/1)      | 0.00%
```

Due to its capacity, test coverage is expected to be 100%.

[N-02] Test arguments and actual contract arguments are incompatible

`FixedPrice.sol.constructor()` test file is `test/FixedPrice.t.sol.setUp()` but used arguments is different. Expected value of arguments should be same.

```
src/minters/FixedPrice.sol:
  42      /// @custom:oz-upgrades-unsafe-allow constructor
  43:     constructor() {
  44:         factory = msg.sender;
  45:         _disableInitializers();
  46:         sale = Sale(0, 0, address(0), type(uint96).max, pa
  47:     }


test/FixedPrice.t.sol:
  11
  12:     function setUp() public virtual override {
  13:         super.setUp();
  14:         fixedSales = new FixedPriceFactory();
  15:         fixedSale = FixedPrice.Sale({
  16:             currentId: uint48(0),    // Same
  17:             finalId: uint48(10),       // Not same
  18:             edition: address(edition), // Same
  19:             price: uint96(uint256(1 ether)),  // Not Same
  20:             saleReceiver: payable(address(69)), // Not San
  21:             startTime: uint96(block.timestamp) // Not Same
  22:         });
  23:     }
```

[N-03] Constant values such as a call to `keccak256()`, should used to immutable rather than constant

There is a difference between constant variables and immutable variables, and they should each be used in their appropriate contexts.

While it doesn't save any gas because the compiler knows that developers often make this mistake, it's still best to use the right tool for the task at hand.

Constants should be used for literal values written into the code, and immutable variables should be used for expressions, or values calculated in, or passed into the constructor.

```
4 results 2 files:

src/Escher.sol:
  11:     bytes32 public constant CREATOR_ROLE = keccak256("CREA
  13:     bytes32 public constant CURATOR_ROLE = keccak256("CURA

src/Escher721.sol:
  17:     bytes32 public constant URI_SETTER_ROLE = keccak256("U
  19:     bytes32 public constant MINTER_ROLE = keccak256("MINTE
```

[N-04] For functions, follow Solidity standard naming conventions

```
src/uris/Generative.sol:
  26
  27:     function tokenToSeed(uint256 _tokenId) internal view r
```

The above codes don't follow Solidity's standard naming convention,

internal and private functions: the mixedCase format starting with an underscore (_mixedCase starting with an underscore)

[N-05] `Function writing` that does not comply with the `Solidity Style Guide`

Order of Functions; ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. But there are contracts in the project that do not comply with this.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html

Functions should be grouped according to their visibility and ordered:

constructor

receive function (if exists)

fallback function (if exists)

external

public

internal

private

within a grouping, place the view and pure functions last

[N-06] Lack of event emission after critical `initialize()` function

To record the init parameters for off-chain monitoring and transparency reasons, please consider emitting an event after the `initialize()` function:

```
1 result 1 file

src/Escher721.sol:
  32:     function initialize(
  33:         address _creator,
  34:         address _uri,
```

```
35:         string memory _name,
36:         string memory _symbol
37:     ) public initializer {
38:         __ERC721_init(_name, _symbol);
39:         __AccessControl_init();
40:
41:         tokenUriDelegate = _uri;
42:
43:         _grantRole(DEFAULT_ADMIN_ROLE, _creator);
44:         _grantRole(MINTER_ROLE, _creator);
45:         _grantRole(URI_SETTER_ROLE, _creator);
46:     }
```

[N-07] Add a timelock to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate (less risk of a malicious owner making a sandwich attack on a user).

Consider adding a timelock to:

```
3 results 3 files

src/minters/FixedPriceFactory.sol:
  47      // @audit Alice Bob , çok yüksek fee durumunda frontru
  48:     function setFeeReceiver(address payable fees) public c
  49:         feeReceiver = fees;
  50      }

src/minters/LPDAFactory.sol:
  45      /// @param fees the address to receive fees
  46:     function setFeeReceiver(address payable fees) public c
  47:         feeReceiver = fees;
  48      }

src/minters/OpenEditionFactory.sol:
  41      /// @param fees the address to receive fees
  42:     function setFeeReceiver(address payable fees) public c
  43:         feeReceiver = fees;
```

[N-08] `Empty blocks` should be *removed* or *Emit* something

Code contains empty block

```
1 result 1 file

src/Escher721.sol:
  25:     constructor() {}
```

Recommended Mitigation Steps

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting.

[N-09] NatSpec comments should be increased in contracts

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation.

In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.
https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommended Mitigation Steps

NatSpec comments should be increased in contracts.

[N-10] Floating pragma

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

https://swcregistry.io/docs/SWC-103

Floating Pragma List: pragma ^0.8.17 (all contracts)

Recommended Mitigation Steps

Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

[N-11] Add parameter to Event-Emit

Some event-emit description don't have a parameter. Add a parameter for front-end website or client app, they can know that something has happened on the blockchain.

```
3 results  3 files

src/minters/FixedPriceFactory.sol:
  48:     function setFeeReceiver(address payable fees) public o
  49          feeReceiver = fees;

src/minters/LPDAFactory.sol:
  45      /// @param fees the address to receive fees
  46:     function setFeeReceiver(address payable fees) public o
  47          feeReceiver = fees;

src/minters/OpenEditionFactory.sol:
  41      /// @param fees the address to receive fees
  42:     function setFeeReceiver(address payable fees) public o
  43          feeReceiver = fees;
```

[N-12] Omissions in Events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts.

However, some events are missing important parameters.

The events should include the new value and old value where possible:

```
3 results  3 files

src/minters/FixedPriceFactory.sol:
```

```
48:     function setFeeReceiver(address payable fees) public c
49         feeReceiver = fees;

src/minters/LPDAFactory.sol:
45      /// @param fees the address to receive fees
46:     function setFeeReceiver(address payable fees) public c
47         feeReceiver = fees;

src/minters/OpenEditionFactory.sol:
41      /// @param fees the address to receive fees
42:     function setFeeReceiver(address payable fees) public c
43         feeReceiver = fees;
```

[N-13] Lack of event emission after critical `initialize()` functions

To record the init parameters for off-chain monitoring and transparency reasons, please consider emitting an event after the `initialize()` functions:

```
src/Escher721.sol:
31      /// @param _symbol the symbol of this contract
32:     function initialize(
33:         address _creator,
34:         address _uri,
35:         string memory _name,
36:         string memory _symbol
37:     ) public initializer {
38:         __ERC721_init(_name, _symbol);
39:         __AccessControl_init();
40:
41:         tokenUriDelegate = _uri;
42:
43:         _grantRole(DEFAULT_ADMIN_ROLE, _creator);
44:         _grantRole(MINTER_ROLE, _creator);
45:         _grantRole(URI_SETTER_ROLE, _creator);
46:     }
```

[N-14] NatSpec is missing

NatSpec is missing for the following functions , constructor and modifier:

```
11 results 6 files

src/Escher721Factory.sol:
  12:     Escher public immutable escher;
  13:     address public immutable implementation;

  17:     constructor(address _escher) {

src/Escher.sol:
  31:     function supportsInterface(

src/Escher721.sol:
  84:     function supportsInterface(

src/minters/FixedPriceFactory.sol:
  12:     address payable public feeReceiver;
  13:     address public immutable implementation;

src/minters/LPDAFactory.sol:
  12:     address payable public feeReceiver;
  13:     address public immutable implementation;

src/minters/OpenEditionFactory.sol:
  12:     address payable public feeReceiver;
  13:     address public immutable implementation;
```

[N-15] Initial value check is missing in Set Functions

```
5 results 4 files
src/minters/FixedPriceFactory.sol:
  48:     function setFeeReceiver(address payable fees) public o
  49          feeReceiver = fees;

src/minters/LPDAFactory.sol:
  46:     function setFeeReceiver(address payable fees) public o
  47          feeReceiver = fees;

src/minters/OpenEditionFactory.sol:
  42:     function setFeeReceiver(address payable fees) public o
  43          feeReceiver = fees;

src/Escher721.sol:
```

```
57:     function updateURIDelegate(address _uriDelegate) publi
58:         tokenUriDelegate = _uriDelegate;

64:     function setDefaultRoyalty(
65:         address receiver,
66:         uint96 feeNumerator
67:     ) public onlyRole(DEFAULT_ADMIN_ROLE) {
68:         _setDefaultRoyalty(receiver, feeNumerator);
69      }
```

Checking whether the current value and the new value are the same should be added.

[N-16] Protect your NFT from copying in POW forks

Ethereum has performed the long-awaited “merge” that will dramatically reduce the environmental impact of the network.

There may be forked versions of Ethereum, which could cause confusion and lead to scams as duplicated NFT assets enter the market.

If the Ethereum Merge, which took place in September 2022, results in the Blockchain splitting into two Blockchains due to the ‘THE DAO’ attack in 2016, this could result in duplication of immutable tokens (NFTs).

In any case, duplicate NFTs will exist due to the ETH proof-of-work chain and other potential forks, and there’s likely to be some level of confusion around which assets are ‘official’ or ‘authentic.’

Even so, there could be a frenzy for these copies, as NFT owners attempt to flip the proof-of-work versions of their valuable tokens.

As ETHPOW and any other forks spin off of the Ethereum mainnet, they will yield duplicate versions of Ethereum’s NFTs. An NFT is simply a blockchain token, and it can work as a deed of ownership to digital items like artwork and collectibles. A forked Ethereum chain will thus have duplicated deeds that point to the same tokenURI.

About Merge Replay Attack:
https://twitter.com/elerium115/status/1558471934924431363?s=20&t=RRheaYJwo-GmSnePwofgag

The most important part here is NFTs tokenURI detail.

If the update I suggested below is not done, duplicate NFTs will appear as a result, potentially leading to confusion and scams.

```
src/Escher721.sol:
  77      /// @param _tokenId the token ID to get the URI for
  78:     function tokenURI(
  79          uint256 _tokenId
```

Recommendation code:

```
src/Escher721.sol:
  77      /// @param _tokenId the token ID to get the URI for
  78:     function tokenURI(
              if(block.chainid != 1) {
                  revert();
              }
  79          uint256 _tokenId
```

Suggestions Summary

Number	Suggestion Details	
[S-01]	Mark visibility of initialize(...) functions as `external`	
[S-02]	Generate perfect code headers every time	

Total: 2 suggestions

[S-01] Mark visibility of initialize(...) functions as `external`

If someone wants to extend via inheritance, it might make more sense that the overridden initialize(...) function calls the internal {...}_init function, not the parent public initialize(...) function.

External instead of public would give more of the sense of the initialize(...) functions to behave like a constructor (only called on deployment, so should only be called externally).

From a security point of view, it might be safer so that it cannot be called internally by accident in the child contract.

It might cost a bit less gas to use external over public.

It is possible to override a function from external to public (= “opening it up”) ✅

But it is not possible to override a function from public to external (= “narrow it down”). ❌

For above reasons you can change initialize(...) to external.

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3750

[S-02] Generate perfect code headers every time

I recommend using header for Solidity code layout and readability

https://github.com/transmissions11/headers

```
/*////////////////////////////////////////////////////////////////
                            TESTING 123
////////////////////////////////////////////////////////////////*/
```

berndartmueller (judge) commented:

> Very good report!

Gas Optimizations

For this contest, 17 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **slvDev** received the top score from the judge.

The following wardens also submitted reports: cryptostellar5, c3phas, tnevler, sakshamguruji, adriro, ajtra, Diana, pfapostol, 0x4non, zaskoh, ReyAdmirado, Dinesh11G, RaymondFam, nicobevi, Bnke0x0, *and* ahmedov .

Gas Optimizations Summary

	Issue	Instances	Saved Gas	Overall gas change from forge snapshot —diff
[G-01]	Refactor Sale struct to avoid using unnecessary storage slot	3	73328	972780
[G-02]	Cache repeated parameters from Sale in buy() function	2	409	10066
[G-03]	Use cached value from memory rather than read storage	1	115	1380
[G-04]	Removing cache Sale to memory saves gas	1	254	6775
[G-05]	Increment in the for loop's postcondition can be made unchecked{++i}/unchecked{i++}	3	215	5855
[G-06]	Simplify / Refactor _end function	2	325	5460
[G-07]	Change for loop behavior by removing add (+1) and ++x is more gas efficient	3	304	4025
[G-08]	+= cost more gas than = +	1	42	756

Gas Report

Total gas saved: **74992**

Overall gas change from forge snapshot: **1002920**

```
test_RevertsWhenEnded_Buy() (gas: -26155 (-3.807%))
test_SellsOut_Buy() (gas: -26157 (-3.824%))
test_LPDA() (gas: -26988 (-3.850%))
test_RevertsWhenSoldOut_Buy() (gas: -27100 (-3.887%))
test_RevertsWhenMintedOut_Buy() (gas: -26564 (-4.306%))
test_WhenMintsOut_Buy() (gas: -26256 (-4.318%))
test_Refund() (gas: -25377 (-6.436%))
```

```
test_WhenNotOver_Refund() (gas: -25377 (-6.436%))
test_RevertsWhenAlreadyRefunded_Refund() (gas: -25731 (-6.472%))
test_RevertsWhenTooSoon_Buy() (gas: -25463 (-6.477%))
test_RevertsWhenTooLate_Cancel() (gas: -25002 (-6.481%))
test_RevertsWhenNotOwner_Cancel() (gas: -25007 (-6.483%))
test_RevertsWhenNoRefund_Refund() (gas: -25377 (-6.507%))
test_Buy() (gas: -25007 (-6.560%))
test_RevertsWhenTooLittleValue_Buy() (gas: -25818 (-6.667%))
test_WhenEnded_Finalize() (gas: -24907 (-7.069%))
test_RevertsWhenNotAdmin_CreateSale() (gas: -2025 (-7.794%))
test_RevertsWhenTooSoon_Buy() (gas: -24797 (-7.880%))
test_RevertsWhenEnded_Buy() (gas: -24813 (-7.886%))
test_RevertsTooMuchValue_Buy() (gas: -24924 (-7.926%))
test_RevertsTooMuchValue_Buy() (gas: -25137 (-8.003%))
test_RevertsWhenTooMany_Buy() (gas: -25155 (-8.006%))
test_RevertsWhenNotOwner_TransferOwnership() (gas: -24708 (-8.01
test_RevertsWhenTooSoon_Buy() (gas: -25239 (-8.025%))
test_RevertsWhenNotOwner_TransferOwnership() (gas: -24758 (-8.03
test_RevertsWhenNotOwner_Cancel() (gas: -24708 (-8.035%))
test_RevertsWhenTooLate_Cancel() (gas: -24720 (-8.040%))
test_RevertsWhenTooLate_Cancel() (gas: -24864 (-8.091%))
test_RevertsWhenTooLittleValue_Buy() (gas: -24924 (-8.096%))
test_RevertsWhenNotOwner_Cancel() (gas: -24869 (-8.104%))
test_Buy() (gas: -24708 (-8.156%))
test_RevertsWhenNotEnded_Finalize() (gas: -25144 (-8.162%))
test_RevertsWhenTooLittleValue_Buy() (gas: -25137 (-8.176%))
test_Buy() (gas: -24847 (-8.207%))
test_RevertsWhenNotAdmin_CreateSale() (gas: -2078 (-8.743%))
test_RevertsWhenNotAdmin_CreateSale() (gas: -2093 (-8.814%))
test_Cancel() (gas: -24400 (-9.382%))
test_Cancel() (gas: -24208 (-9.592%))
test_Cancel() (gas: -24699 (-10.928%))
test_RevertsWhenInitialized_Initialize() (gas: -1971 (-10.952%))
test_CreateSale() (gas: -24216 (-11.069%))
test_RevertsWhenInitialized_Initialize() (gas: -2137 (-11.845%))
test_CreateSale() (gas: -24277 (-12.900%))
test_CreateSale() (gas: -24316 (-12.913%))
test_RevertsWhenInitialized_Buy() (gas: -2089 (-13.919%))
test_RevertsWhenInitialized_Buy() (gas: -4392 (-29.262%))
test_RevertsWhenInitialized_Finalize() (gas: -4414 (-29.460%))
Overall gas change: -1002920 (-413.087%)
```

[G-01] Refactor Sale struct to avoid using unnecessary storage slot

We can use `uint32` for time variables instead of `uint96` since it will be enough until the year **2106** but can save a decent amount of gas.

```
    file: src/minters/LPDA.sol

    struct Sale {
                    // slot 1
                    uint48 currentId;
                    uint48 finalId;
                    address edition;
                    // slot 2
                    uint80 startPrice;
                    uint80 finalPrice;
    +   uint32 startTime;
    +   uint32 endTime;
    -               uint80 dropPerSecond;
                    // slot 3
    -               uint96 endTime;
    +   uint80 dropPerSecond;
                    address payable saleReceiver;
    -               // slot 4
    -               uint96 startTime;
    }
```

```
test_Buy() (gas: -24451 (-6.415%))

Overall gas change: -370656 (-99.469%)
```

Same here, we can use `uint32` for the time variables

```
    file: src/minters/OpenEdition.sol

    struct Sale {
              // slot 1
              uint72 price;
              uint24 currentId;
              address edition;
              // slot 2
    -         uint96 startTime;
    +         uint32 startTime;
    -   uint96 endTime;
```

```
    +   uint32 endTime;
              address payable saleReceiver;
    -         // slot 3
    -         uint96 endTime;
    }


test_Buy() (gas: -24420 (-8.061%))

Overall gas change: -301789 (-149.642%)
```

In FixedPrice.sol we use `uint32` for the `startTime` variable and `uint32` for `currentId` and `finalId` . Since in OpenEdition.sol use `uint24` for `currentId`

```
    file: src/minters/FixedPrice.sol

    struct Sale {
              // slot 1
    -         uint48 currentId;
    -         uint48 finalId;
    +   uint32 currentId;
    +         uint32 finalId;
    +   uint32 startTime;
              address edition;
              // slot 2
              uint96 price;
              address payable saleReceiver;
    -         // slot 3
    -         uint96 startTime;
    }


test_Buy() (gas: -24457 (-8.078%))

Overall gas change: -300344 (-126.975%)
```

[G-02] Cache repeated parameters from Sale in `buy()` function.

Instead of caching the entire Sale object, only cache the `currentId` and `finalId` parameters in the function. This will save on gas consumption and improve

efficiency.

```
file: src/minters/FixedPrice.sol

function buy(uint256 _amount) external payable {
-         Sale memory sale_ = sale;
+         uint48 currentId = sale.currentId;
+         uint48 finalId = sale.finalId;
-         IEscher721 nft = IEscher721(sale.edition);
+         IEscher721 nft = IEscher721(sale.edition);
-         require(block.timestamp >= sale_.startTime, "TOO SOON")
+         require(block.timestamp >= sale.startTime, "TOO SOON");
-         require(_amount * sale_.price == msg.value, "WRONG PRIC
+         require(_amount * sale.price == msg.value, "WRONG PRICE
-         uint48 newId = uint48(_amount) + sale_.currentId;
+         uint48 newId = uint48(_amount) + currentId;
-         require(newId <= sale_.finalId, "TOO MANY");
+         require(newId <= finalId, "TOO MANY");

-         for (uint48 x = sale_.currentId + 1; x <= newId; x++) {
+         for (uint48 x = currentId + 1; x <= newId; x++) {
              nft.mint(msg.sender, x);
          }

          sale.currentId = newId;

          emit Buy(msg.sender, _amount, msg.value, sale);

-         if (newId == sale_.finalId) _end(sale);
+         if (newId == finalId) _end(sale);
      }
```

```
test_Buy() (gas: -189 (-0.062%))
```

```
file: src/minters/LPDA.sol

function buy(uint256 _amount) external payable {
     uint48 amount = uint48(_amount);
-     Sale memory temp = sale;
+     uint48 currentId = sale.currentId;
+     uint48 finalId = sale.finalId;
-     IEscher721 nft = IEscher721(temp.edition);
+     IEscher721 nft = IEscher721(sale.edition);
```

```
-     require(block.timestamp >= temp.startTime, "TOO SOON");
+     require(block.timestamp >= sale.startTime, "TOO SOON");
      uint256 price = getPrice();
      require(msg.value >= amount * price, "WRONG PRICE");

      amountSold += amount;
-     uint48 newId = amount + temp.currentId;
+     uint48 newId = amount + currentId;
-     require(newId <= temp.finalId, "TOO MANY");
+     require(newId <= finalId, "TOO MANY");

      receipts[msg.sender].amount += amount;
      receipts[msg.sender].balance += uint80(msg.value);

-     for (uint256 x = temp.currentId + 1; x <= newId; x++) {
+     for (uint256 x = currentId + 1; x <= newId; x++) {
          nft.mint(msg.sender, x);
      }

      sale.currentId = newId;

-     emit Buy(msg.sender, amount, msg.value, temp);
+     emit Buy(msg.sender, amount, msg.value, sale);
-     if (newId == temp.finalId) {
+     if (newId == finalId) {
          sale.finalPrice = uint80(price);
          uint256 totalSale = price * amountSold;
          uint256 fee = totalSale / 20;
          ISaleFactory(factory).feeReceiver().transfer(fee);
-         temp.saleReceiver.transfer(totalSale - fee);
+         sale.saleReceiver.transfer(totalSale - fee);
          _end();
      }
  }
```

`test_Buy() (gas: -220 (-0.058%))` and `test_LPDA() (gas: -319 (-0.046%))`

[G-03] Use cached value from memory rather than read storage

file: src/minters/OpenEdition.sol

```
    function buy(uint256 _amount) external payable {
                    uint24 amount = uint24(_amount);
                    Sale memory temp = sale;
                    IEscher721 nft = IEscher721(temp.edition);
                    require(block.timestamp >= temp.startTime, "TOO
                    require(block.timestamp < temp.endTime, "TOO LAT
    -               require(amount * sale.price == msg.value, "WRONG
    +               require(amount * temp.price == msg.value, "WRONG
                    uint24 newId = amount + temp.currentId;

                    for (uint24 x = temp.currentId + 1; x <= newId;
                        nft.mint(msg.sender, x);
                    }
                    sale.currentId = newId;

                    emit Buy(msg.sender, amount, msg.value, temp);
    }
```

```
test_Buy() (gas: -115 (-0.038%))
```

[G-04] Removing cache Sale to memory saves gas

it will save gas in `buy()` and `refund()` functions

```
    file: src/minters/LPDA.sol

    function getPrice() public view returns (uint256) {
    -               Sale memory temp = sale;
    -               (uint256 start, uint256 end) = (temp.startTime,
    +               (uint256 start, uint256 end) = (sale.startTime,
                    if (block.timestamp < start) return type(uint256
    -               if (temp.currentId == temp.finalId) return temp.
    +               if (sale.currentId == sale.finalId) return temp.

    -               uint256 timeElapsed = end > block.timestamp ? bl
    -               uint256 timeElapsed = end > block.timestamp ? bl
    -               return temp.startPrice - (temp.dropPerSecond * t
    +   return sale.startPrice - (sale.dropPerSecond * timeElapsed);
    }
```

`test_Buy() (gas: -254 (-0.067%))` and `test_Refund() (gas: -507 (-0.129%))`

[G-05] Increment in the for loop's post condition can be made unchecked

The `newId` variable is calculated using addition, which means it cannot overflow. (In Solidity version 0.8.0 and higher)

```
file: src/minters/FixedPrice.sol

function buy(uint256 _amount) external payable {
                ...

-     for (uint48 x = sale_.currentId + 1; x <= newId; x++) {
+     for (uint48 x = sale_.currentId + 1; x <= newId; ) {
         nft.mint(msg.sender, x);
+        unchecked {
+             x++;
+        }
      }
                ...
}
```

`test_Buy() (gas: -78 (-0.026%))`

```
file: src/minters/LPDA.sol

function buy(uint256 _amount) external payable {
                ...

-     for (uint48 x = sale_.currentId + 1; x <= newId; x++) {
+     for (uint48 x = sale_.currentId + 1; x <= newId; ) {
         nft.mint(msg.sender, x);
+        unchecked {
+             x++;
+        }
      }
                ...
}
```

`test_Buy() (gas: -59 (-0.015%))` and `test_LPDA() (gas: -590 (-0.084%))`

```
file: src/minters/OpenEdition.sol

function buy(uint256 _amount) external payable {
                ...

-    for (uint48 x = sale_.currentId + 1; x <= newId; x++) {
+    for (uint48 x = sale_.currentId + 1; x <= newId; ) {
        nft.mint(msg.sender, x);
+       unchecked {
+            x++;
+       }
    }
                ...
}
```

`test_Buy() (gas: -78 (-0.026%))`

[G-06] Simplify `_end` function

Also in OpenEdition.sol we can remove unnecessary struct cache in `finalize()` function.

```
file: src/minters/OpenEdition.sol

function cancel() external onlyOwner {
    require(block.timestamp < sale.startTime, "TOO LATE");
-   _end(sale);
+   _end();
}

function finalize() public {
-   Sale memory temp = sale;
-   require(block.timestamp >= temp.endTime, "TOO SOON");
+   require(block.timestamp >= sale.endTime, "TOO SOON");
    ISaleFactory(factory).feeReceiver().transfer(address(this).b
-   _end(temp);
+   _end();
}
```

```
- function _end(Sale memory _sale) internal {
+ function _end() internal {
-           emit End(_sale);
+           emit End(sale);
-           selfdestruct(_sale.saleReceiver);
+           selfdestruct(sale.saleReceiver);
        }
```

```
test_Cancel() (gas: -304 (-0.135%))
```

In FixedPrice.sol refactor `_end` function save 21 gas.

```
file: src/minters/FixedPrice.sol

function cancel() external onlyOwner {
    require(block.timestamp < sale.startTime, "TOO LATE");
-   _end(sale);
+   _end();
}

function buy(uint256 _amount) external payable {
...
-               if (newId == sale_.finalId) _end(sale);
+               if (newId == sale_.finalId) _end();
}

- function _end(Sale memory _sale) internal {
+ function _end() internal {
+     Sale memory temp = sale;
-           emit End(_sale);
+     emit End(temp);
            ISaleFactory(factory).feeReceiver().transfer(address
-           selfdestruct(_sale.saleReceiver);
+    selfdestruct(temp.saleReceiver);
        }
```

```
test_Cancel() (gas: -21 (-0.008%))
```

[G-07] Change for loop behavior by removing add (+1) and ++x is more gas efficient

file: src/minters/FixedPrice.sol

```
function buy(uint256 _amount) external payable {
                                    ...
-       for (uint48 x = sale_.currentId + 1; x <= newId; x++) {
+       for (uint48 x = sale_.currentId; x < newId; ++x) {
            nft.mint(msg.sender, x);
        }
                                    ...
    }
```

`test_Buy() (gas: -105 (-0.035%))`

file: src/minters/LPDA.sol

```
function buy(uint256 _amount) external payable {
                                    ...
-       for (uint256 x = temp.currentId + 1; x <= newId; x++) {
+       for (uint256 x = temp.currentId; x < newId; ++x) {
            nft.mint(msg.sender, x);
        }

                                    ...
    }
```

`test_Buy() (gas: -97 (-0.025%))` and `test_LPDA() (gas: -210 (-0.030%))`

file: src/minters/OpenEdition.sol

```
function buy(uint256 _amount) external payable {
                ...
-   for (uint24 x = temp.currentId + 1; x <= newId; x++) {
+   for (uint24 x = temp.currentId; x < newId; ++x) {
        nft.mint(msg.sender, x);
    }

                ...
```

```
    }
```

```
test_Buy() (gas: -102 (-0.034%))
```

[G-08] <x> += <y> cost more gas than <x> = <x> + <y>

```
    file: src/minters/LPDA.sol

    function buy(uint256 _amount) external payable {
                    ...
                    require(msg.value >= amount * price, "WRONG PRIC

    -               amountSold += amount;
    +               amountSold = amountSold + amount;
                    uint48 newId = amount + temp.currentId;

                    ...
    }
```

```
test_Buy() (gas: -42 (-0.011%))
```

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top