NFT X | Contracts

R E A U D I T

Distributed Lab

By Artem Chystiakov and Armen Arakelian, APRIL 2022

Summary: production-ready.

During the audit, there were no critical and one high issues found. The overall code quality is great and it is fairly easy to navigate through. The design and architecture are decent, however, the marketplace misses several key features that would be great to have (see Afterthoughts section). Even though the code itself is quite secure, all the described issues are recommended to be properly addressed.

The reaudit of the provided contracts has been made and all of the highlighted vulnerabilities during the former audit session have been either directly fixed or addressed in way that neglect their severity. We think that the project is ready to go live.

The audit is made regarding these commit hashes:

Core contracts: d886dd35b7d8e9bc259399fe62e41aee6ae73427
Lootbox contracts: 7f7b05ec3ef15e1d5c255bffe972724f59977fb2

The reaudit is made considering the following commit hashes:

Core contracts: 0f3150981e1ce45428b2cd1ec1e3663c4c2e96c6
Lootbox contracts: c7eb931a90042d505a9661710bbcc0a1ff71feee

STRUCTURE AND ORGANIZATION OF DOCUMENT

The information about the severity of the bugs is given below. The list starts with the description of the critical bugs in red and ends with the informational ones in blue.

crit

Critical - The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

high

High - The issue affects the ability of the contract to compile or operate in a significant way.

med

Medium - The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

low

Low - The issue has minimal impact on the contract’s ability to operate.

info

Informational - The issue has no impact on the contract’s ability to operate.

CORE CONTRACTS

01 TOKENTRADER.SOL

The `buy()` function can be reentered

The `buy()` function can be reentered if `ERC721` or `ERC1155` asset is bought. The reentrancy is possible due to “safe” functions callbacks. These “safe” functions call the token receiver address to validate that a certain token has been received. As a result, giving the execution control to the receiver, who would call the `buy()` function sequentially.

In the case of `ERC721` asset, this exploit breaks the contract’s logic by updating the recipient’s balance more frequently than the total supply value. This leads to wrong `balanceOf()` outputs, which might even be greater than the `totalSupply()`.

```
function _safeMint(
      address to,
      uint256 quantity,
      bytes memory _data
) internal virtual {
      . . .

      uint256 startTokenId = currentIndex;

      AddressData memory addressData = _addressData[to];
      _addressData[to] = AddressData(
            addressData.balance + uint128(quantity),
            addressData.numberMinted + uint128(quantity)
      );
      _ownerships[startTokenId] = TokenOwnership(to,
      uint64(block.timestamp));

      uint256 updatedIndex = startTokenId;

      for (uint256 i = 0; i < quantity; i++) {
            emit Transfer(address(0), to, updatedIndex);
            require(
                  _checkOnERC721Received(address(0), to,
                  updatedIndex, _data),
                  "ERC721A: transfer to non ERC721Receiver
                  implementer"
            );
            updatedIndex++;
      }

      currentIndex = updatedIndex;
      _afterTokenTransfers(address(0), to, startTokenId,
      quantity);
}
```

01 TOKENTRADER.SOL

Reentrancy example: if the reentrancy happens twice (the recipient reenters `buy()` function twice) in the `_checkOnERC721Received()` callback, the `balance` and `numberMinted` variables will be updated 3 times, however, the `currentIndex` will get updated only once, resulting in the discrepancy between data.

Recommendation:

Remove the lines `uint256 updatedIndex = startTokenId;` and `currentIndex = updatedIndex;` from the code. Change the loop implementation to:

```
for (uint256 i = 0; i < quantity; i++) {
      uint256 updatedIndex = currentIndex++;

      emit Transfer(address(0), to, updatedIndex);
      require(
            _checkOnERC721Received(address(0), to, updatedIndex,
            _data),
            "ERC721A: transfer to non ERC721Receiver
            implementer"
      );
}
```

Add the `nonReentrant` modifier to the `buy()` function.

01 TOKENTRADER.SOL

The prices for the entire ERC1155 collection are the same

The ERC1155 token standard is made to maintain multiple tokens in a single contract. These can be either NFTs or SFTs (semi fungible tokens). Usually, these inner tokens mean completely different things and have completely different metadata assigned to them, including different prices. However, the buy() function is made in a way that makes every inner token of ERC1155 standard priced the same.

```
function buy(LibAsset.Asset memory asset) external payable {
      . . .

      if (asset.assetType == LibAsset.AssetType.ERC721) {
            . . .
      } else if (asset.assetType == LibAsset.AssetType.ERC1155) {
            require(
                  msg.value >= tradeInfo[asset.token].price *
                  asset.amount,
                  "TokenTrader: message value too low"
            );

            IERC1155Mintable(asset.token).mint(msg.sender,
            asset.id, asset.amount);
      }

      . . .
}
```

Recommendation:

Rewrite the function’s logic in a way that would take into account different prices for different ERC1155 token ids.

02 TOKENTRADER.SOL

Double-check the `initializer` modifier

If you were to deploy the `Exchange` contract proxy manually and called `initialize()` method after deployment there, the execution would revert. This is due to the `initializer` modifier used in the `__TokenTrader_init()` function instead of the `onlyInitializing` one.

Recommendation:

Double-check that the proxy is always deployed with the additional `constructor` data.

The contract is too greedy

In the `buy()` function, the correctness of `msg.value` is checked with `">="` sign, which might cause leakage of users `ether`. Basically, the contract might take more `ether` than required and not return the remainder back.

Recommendation:

Either change the comparison sign to `"=="` or send back the `ether` remainder. In case of using `address.call{value: <smth>}("")`, do not forget to protect against possible reentrancies.

02 EXCHANGE.SOL

The payment token can be an arbitrary `ERC20` token

This is a potential security concern because the precious NFT asset might be sold for a useless yet malicious `ERC20` token. The token itself might experience high volatility or at the worst – rug pulled.

Recommendation:

Reassure that the payment method is either checked against the whitelisted token addresses on the contracts side or limit the UI to only allow the order signatures with the specific payment tokens. The latter option is not ideal, but it will protect 99% of users.

The `OwnableUpgredeable` contract is inherited twice

The `Exchange` contract inherits `OwnableUpgradeable` and `TokenTrader` contracts, whilst `TokenTrader` inherits `OwnableUpgradeable` itself. The code does not introduce any bugs, yet there is no need for such explicitness.

```
contract Exchange is OwnableUpgradeable, TokenTrader {
      . . .
}
```

Recommendation:

Remove `OwnableUpgradeable` for the inheritance list.

03 LIBORDER.SOL

Order `nonce` might not be enough

There is a `nonce` value assigned to each created order on the platform. We assume that `nonce` is required to correctly process similar orders and is fully maintained by the backend. Because the `nonce` is only a `uint8` variable that can only store values between `0` and `255`, the users are only able to place the same orders `255` times maximum. The case might be worse if the user goes in a loop by placing an order and then canceling it, reaching the `nonce` limit. It is not an exception that this tiny limit might be reached naturally.

```
struct Order {
        address account;
        address taker;
        OrderSide side;
        LibAsset.Asset commodity;
        LibAsset.Asset payment;
        uint64 expiry;
        uint8 nonce;
        bytes permitSig;
        LibSig.Signature orderSig;
}
```

Recommendation:

Change the `nonce` type to `uint256`. Because the `nonce` value gets never stored in the contract, this change will not affect the execution cost.

LOOTBOX CONTRACTS

01 VRFCONSUMERBASEUPGRADEABLE.SOL

The `initializer` modifier

The contract was redesigned from the original chainlink consumer contracts to be upgradeable and the `__VRFConsumerBase_init` method was added. However, since the `VRFConsumerBaseUpgradeable` contract is abstract, the `initializer` modifier has to be changed to `onlyInitializing` one.

```
function __VRFConsumerBase_init(address _vrfCoordinator, address _link)
      public
      initializer
{
      . . .
}
```

Recommendation:

Change the modifier to `onlyInitializing`.

02 LOOTBOX.SOL

The algorithm of fulfilling and claiming the box can be optimized significantly

Right now the algorithm works in the following way: the `fulfillOpenBox()` function accepts the `content` array and simply copies provided weights into the contract's storage. The algorithm also summarizes all the weights in a separate variable to further crop the given `VRF` random value. Then the user calls `claimBox()` function, loops through all the stored weights, and sums them up until the answer exceeds the cropped `VRF` value. This algorithm is not at all optimal.

Recommendation:

The better implementation would be:

1. Store prefix summed weights array instead of plain weights array in the `fulfillOpenBox()` function.

2. Remove the `totalWeight` value completely (the last element of the prefix sum array can be used instead).

3. Do a binary search (`upperBound`) on the prefix summed weights array to find the exact spot that is greater than the cropped `VRF` value - the index of than value will be the desired answer. Openzeppelin `Arrays` library `findUpperBound` method could be used.

02 LOOTBOX.SOL

The `fulfillRandomness()` function might revert

The `revert` of the `fulfillRandomness()` function might happen if the `totalWeight` value is zero. The only way this value can be assigned to zero is if the provided weights sum up to zero. There is no check performed on the `totalWeight` value in the `fulfillOpenBox()` function body, so there is a slim chance of setting `totalWeight` variable to zero. And if the `fulfillRandomness()` function eventually reverts, the chainlink `VRF` oracle will stop serving the contract.

Recommendation:

Add the `require` statement to the `fulfillOpenBox()` function that ensures that `totalWeight` is greater than zero.

AFTERTHOUGHTS

- The common code of the `EIP712` domain typehash and `corresponding _hashDomain()` function might be moved to a separate contract and reused in both `ERC721Permit` and `ERC1155Permit` contracts.

- ~~There are several “magic numbers” used in the `LibOrder` library that should be declared as `constants`.~~

- The `internal _setOwnersExcplicit()` function in the `ERC721A` contract is never used.

- ~~Contracts are unable to place orders themselves due to the code always requiring the order placer signature to be present. For example, the wyvern protocol (opensea smartcontracts solution) checks if the given order is approved (`mapping` of order’s `hash` to `bool`) and if it is the case, does not proceed with the signature validation. This allows contracts to approve their orders before the matching.~~

- ~~The order is missing “`startTime`” feature. It would be extremely useful to be able to create future orders beforehand, just like on Opensea. For example, this feature would enable launching planned NFT sales.~~

With great appreciation and respect.

Distributed Lab

APRIL 2022