Known Origin Digital Asset v3 Audit

June 2021

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the Known Origin Digital Asset v3 project. First we will provide a summary of our discoveries and then we will show the details of our findings.

Summary

The contracts audited are from the Known Origin Next Generation Project repository at this link (https://github.com/knownorigin/known-origin-contracts-next-gen). The audit started based on the commit bcbd6b7bfe7e69ef44cb7d1ed524ca810a86d2dc, and was updated to reflect changes based on this second commit: bede703e34f5ca4878cc5dcebc7a2f9fa649b189.

The project includes several contracts (listed below) which define a token and a marketplace to handle the sales of these tokens. Four types of listing types are supported: Buy now, Stepped auctions, Reserve auctions, and Offers.

Contracts

The audited contracts are:

- core/BaseKoda.sol: Basic abstract contract with basic token functionality
- core/KnownOiginDigitalAssetV3.sol: Main token contract
- core/Konstants.sol
- core/IERC721Ownable.sol
- core/IERC2309.sol
- core/IERC2981.sol
- core/composable/TopDownERC20Composable.sol: abstract contract defining token.
- marketplace/BaseMarketplace.sol: abstract contract with basic marketplace functionality
- marketplace/KODAV3PrimaryMarketplace.sol: Contract handling primary sales

- marketplace/KODAV3SecondaryMarketplace.sol: Contract handling secondary sales
- marketplace/BuyNowMarketplace.sol: Contract handline on-the-moment sales.
- marketplace/ReserveAuctionMarketplace.sol: Contract handling Auctions.

Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.

- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED
Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Findings and Fixes

The table below summarizes findings. Details follow in the next section.

ID	Title	Severity	Status
ME-01	Insufficient Input Validation	Medium	Fixed
ME-02	Bid Deniability in placeTokenBid() Function	Medium	Fixed
MI-01	Lost Offers Due To Faulty Design	Minor	Expected behavior
MI-02	Bidder with Best Bid May Be Ignored	Minor	Expected behavior
MI-03	Front-running Attack on acceptTokenBid()	Minor	Expected behavior
MI-04	Function Enabling Arbitrary Callers Leads to Auction Deniability	Minor	Fixed
MI-05	Repeated Structures May Lead to Data Mismatch	Minor	Business rules apply

Issues Found by Severity

Critical severity

No issues found.

Medium severity

ME-01 Insufficient Input Validation

The function `isEditionSoldOut` in the `KnownOriginDigitalAssetV3.sol` contract does not check whether the sought `editionId` exists, hence when queried with an inexistent `tokenId` the function returns `true`.

This could lead users to think that an edition exists when it does not and may mislead actions like airdrops or reward mechanisms if the public function `editionExists()` is not used previously.

Recommendation

Add a require statement like in similar functions. For example,

```
require(_editionExists(_editionId), "Edition does not exist");
```

Status

The recommendation was applied and the issue no longer exists.

ME-02 Bid Deniability in placeTokenBid() Function

In `placeTokenBid(uint256 _tokenId)` and `rejectTokenBid(uint256 _tokenId)` functions (repeated in the `KodaV3PrimaryMarketplace.sol` and KodaV3SecondaryMarketplace.`sol`) there are calls to `_refundBidder()` in those cases where an earlier offer was made. Since `_refundBidder()` requires the refund to be successful, a bidder could block others from placing bigger bids by simply refusing the transfer.

The malicious bidder would then win the bid, manipulating the result of the auction.

Recommendation

Decouple bidding from refund (the success of the refund should not be required for a new bid to be placed).

Refunds are placed in a refund queue but the bid is immediately replaced. For example, add a structure to store missed refunds which acts as a queue and a periodic procedure which handles these refunds.

Status
A successful refund is no longer required for someone to outbid an existing bid. The issue no longer exists. Documentation should reflect this.

Minor severity
MI-01 Lost Offers Due To Faulty Design
A drawback on the bidding logic in `KODAV3PrimaryMarketplace.sol` could result in losing valid offers when a sale is not made before the `bidLockupPeriod`.

An example follows: a first buyer (buyer_A) places an offer. As a result, the `editionOffers` mapping is updated as follows:

```
editionOffers[_editionId] = Offer(msg.value,                (1)
                                  _msgSender(),
                                  _getLockupTime());
```

Next, a second buyer (buyer_B) places a second and better offer, and as a result buyer_A is refunded and this second offer replaces offer `(1)` above, thus updating `editionOffers`. If the lockup period has elapsed (`timestamp >= lockupUntil`), then buyer_B may cancel the offer using `withdrawEditionBid()` later and no bid remains.

Recommendation
Allow bidders to optionally maintain their bid, even if it is not the best bid, by storing losing bids until the auction is finished.

Status
Development team claims this is the expected behavior, explaining that the lockout period may be chosen to prevent this, and off-chain events (such as emails) notify owners when new bids are received. These actions do mitigate the issue thus minimizing the risk.

MI-02 Bidder with Best Bid May Be Ignored
When a token creator calls `convertOffersToBuyItNow()` and there already is an offer, the bidder is refunded. This happens even in the case when

```
offer.offer >= _listingPrice
```

in which case, the bidder could be outpaced by another buyer. Moreover, this new buyer could be paying an amount smaller than `offer.offer`! Hence, both the initial bidder and token creator would lose.

Recommendation
Allow bidders to optionally have their bids automatically converted into buys if the creator converts the offer from type 'listing' to 'buy now'.

Status
Development team claims this is the expected behavior, explaining that the owner can change from "offer" to "buy now" at his discretion. Documentation should thus reflect this, so owners can make informed decisions.

MI-03 Front-running Attack on acceptTokenBid()
A malicious bidder could pay a smaller-than-expected price using a front-running attack on the bid-acceptance call. Assume the following situation:

1. a bidder made a winning offer (e.g., through the function `placeTokenBid()`),
2. `_offerPrice` is strictly smaller than `offer.offer`,
3. the `lockupUntil` time has elapsed, and
4. the owner calls `acceptTokenBid()`.

Then, the bidder could successively withdraw his initial offer and make one for `_offerPrice` hence saving `offer.offer - _offerPrice`.

Recommendation
Document the impact of this front running advantage and warn users about this possibility.

Status
Development team claims this is the expected behavior, explaining that an informed owner can select the value `_offerPrice` at his discretion; hence choosing this to be equal to `offer.offer` does eliminate the problem. Documentation should thus reflect this, so owners can make informed decisions.

MI-04 Function Enabling Arbitrary Callers Leads to Auction Deniability
An arbitrary contract may call `listForReserveAuction` on a token if this is an edition of size one.

Moreover, if the token is already listed in a reserve auction, whenever someone wants to call `placeBidOnReserveAuction()`, a malicious user could front-run him

calling `listForReserveAuction` setting a `reserveAuction.startDate` in the future so that the bid is rejected.

Recommendation
This issue would not exist if only token creators are allowed to call this function.

Status
The recommendation was applied and the issue was eliminated.

MI-05 Repeated Structures May Lead to Data Mismatch
The parameters `editionDetails[editionId].uri` may be also obtained through the `tokenUriResolver` interface (`tokenUriResolver.editionURI(editionId)`). In fact, the function editionURI will use the latter if tokenUriResolverActive() returns True or else the former.

Status
Development Team explains that the data mismatch is expected as the token resolver will become the defacto returned data payload once called and set, super seeding any original uri set on creation. The business rule would fix the issue.

Enhancements

EN-01 Possible Reduction in Variable Size
The variable `editionSize` is created as an `uint96`, but it holds integers that are bounded by `MAX_EDITION_SIZE` which is initialized as 1000. We can have `editionSize` to be `uint16` (allowing `MAX_EDITION_SIZE` to grow up to 65k).

Status
Not completely fixed.

EN-02 Untight Packing Leading to Excess Gas Spent
The function `getEditionDetails(uint256 _tokenId)` in the `KnownOriginDigitalAssetV3.sol` contract returns the following values

```
(address _originalCreator, address _owner, uint256 _editionId,
uint256 _size, string memory _uri)
```

of respective sizes 160, 160, 256, (256), 59. These are not tightly packed into 32 byte slots and hence could lead to inefficient gas spends.

Recommendation
Modifying the output so that `_size` is of the smallest possible `uint` and placing `uint256 return values` at the start, reduces gas.

Status
Fixed.

EN-03 Variables with the same name may lead to problems
In both contracts `BaseKoda.sol` and `KodaV3SecondaryMarketplace.sol` a variable named `secondarySaleRoyalty` is defined, with the same objective. In both contracts the value is initialized as

```
uint256 public secondarySaleRoyalty = 12_50000
```

and can be changed only by an admin. However, the admin could call only one of the update functions and the values would fall out of sync.

Status
Development team understands the risks and answered that there are business processes in place to align changes in both variables.

EN-04 Spurious Data in editionDetails
The struct `editionDetails` contains three items: the creator, editionSize and an URI of a token. However, the URI may also be stored in another map which can be accessed through the proxy tokenUriResolver. So, eventually, when the proxy is used for URIs, the `editionDetails` will have outdated information in the URI entry but should be updated in the other two entries. This may lead to synchronization problems.

Status
Development Team explains that the token resolver will become the defacto returned data payload once called and set, super seeding any original uri set on creation. This fixes the issue.

EN-05 Division-by-Zero Not Prevented
The functions updateModulo and updateBasisPointsModulo allow modulus to be set to zero. A ‘`require`‘ statement should prevent this.

Status
The issue has been fixed.

Relevant Observations

Inaccurate Check With Negligible Impact

The implementation of `_safeTransferFrom` uses the code

```
uint256 receiverCodeSize;
assembly {
    receiverCodeSize := extcodesize(_to)
}
if (receiverCodeSize > 0) {...}
```

which may be called by a maliciously-crafted contract that appears to have code of size 0, and therefore the check fails. This check should be used with care even if the security impact of the `to_` being a contract is negligible.

A known old example (a zero-length-code Attacker contract cheating a Victim contract) follows:

```
pragma solidity 0.4.25;

contract Victim {
    function isContract() public view returns(bool){
      uint32 size;
      address a = msg.sender;
      assembly {
        size := extcodesize(a)
      }
      return (size > 0);
    }

}


contract Attacker {

    bool public iTrickedIt;
    Victim v;

    constructor(address _v) public {
        v = Victim(_v);
        // address(this) doesn't have code, yet
        iTrickedIt = !v.isContract();
    }
}
```

Conclusion

The Known Origin project introduces new trading mechanisms on its marketplace which introduce new attack vectors that have not been analyzed for the known mechanisms used in other published marketplaces. Especially in which refers to front-running attacks and correct cryptographic validations.

ME-01 issue is related to the problems caused by behavior users expect from a specific function based on its name. ME-02 issue describes a situation in which bids could get stuck. While there is a function that the owner could call to recover from these situations, the system should not rely on this contingency mechanism.

The minor issues we found are related to a lack of documentation that may lead users to misuse the functions as intended or not be aware of the possible front-running advantages that could take place.

The enhancements we propose are related to reducing gas consumption, preventing a division-by-zero scenario and mitigating the risk of having different values for the variables that are declared with the same name in two different contracts. We also posed a relevant observation regarding the use of an inaccurate method to verify whether a given address is a smart contract or not.

All medium-severity issues have been fixed and minor-severity issues have either been fixed, should be fixed through business rules that are to be applied on-chain, or have been accepted as they are part of the expected behaviour. In that case, documentation and business processes reduce the risk to something that is negligible.