Formal Verification of Zesty Smart Contracts

Summary

This document describes the specification and verification of Zesty Market using the Certora Prover. The work was undertaken from June 21, 2021 to July 4, 2021. All commits are run through the Certora Prover.

The scope of our verification was the Zesty Markets logic.

The Certora Prover proved the implementation of the Zesty Market is correct with respect to the formal rules written by the Zesty and the Certora teams. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of Zesty. All the rules are publically available (https://github.com/zestymarket/zesty-contracts-v2/tree/main/specs).

Certora Prover run results:

- Zesty Market V1_1 (https://vaas-stg.certora.com/output/43260/a03f1d5d9acc92de66e1?anonymousKey=28767ccc25cd5fec3d4108de7e13990cfd9ce1a2)
- Zesty Market V2 (https://vaas-stg.certora.com/output/43260/5167604c74cafa114f69?anonymousKey=6ba5e3192e6b10e14b762c27313ca31a5c6605f6)
- Zesty NFT (https://vaas-stg.certora.com/output/43260/6c39a6b984e26acea5c3/?anonymousKey=895cd4f0d63d2ba65d34b35bace9797d257370ae)

List of Main Issues Discovered

Severity: High

Issue:	**Integrity of auction times can be violated**
Rules Broken:	Time ranges validity
Description:	Contract time start is not checked to be within `(auction time start, contract time end)`. Specifically, it is only checked to be greater than `auction time start`.
Mitigation/Fix:	Add a require statement to auction creation function.

Severity: High

Issue:	**Auction cancel operation may fail unexpectedly**
Rules Broken:	Revert conditions for auction cancellation
Description:	An unexpected subtraction overflow caused by wrong ordering of storage writes. Found by the Zesty team.
Mitigation/Fix:	Reorder storage updates so that we do not dereference deleted storage.

Severity: Recommendation

Issue:	**No need to keep an extra variable for an address**
Rules Broken:	N/A
Description:	There is no need to keep an extra state variable of type address to mirror the value of a type such as `IERC20` .
Mitigation/Fix:	Remove the address state variable.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

✔️indicates the rule is formally verified on the latest reviewed commit. We write ✔️* when the rule was verified on a simplified version of the code (or under some assumptions).

❌ indicates the rule was violated under one of the tested versions of the code.

✍️indicates the rule is not yet formally specified.

🔁indicates the rule is postponed.

We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.

The syntax {p} (C1 ∼ C2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states. As a special case, C1∼op C2, where op is a getter, indicating that C1 and C2 result in states with the same value for op.

Verification of ZestyNFT

Implementation of ERC721 for Zesty NFT tokens.

Functions

1. `balanceOf(address owner): uint256`
   Returns the number of NFTs held by the given owner.

2. `ownerOf(uint256 tokenId): address`
   Returns the owner of the NFT with the given token ID.

3. `tokenOfOwnerByIndex(address owner, uint256 index): uint256`
   Returns the token ID held by the given owner in the provided index.
   (exposes internal state.)

4. `totalSupply(): uint256`
   Returns the number of NFTs in circulation.

5. `tokenByIndex(uint256 index): uint256`
   Returns the token ID in the given global index.
   (exposes internal state.)

6. `mint(string uri)`
   Mints a new NFT.

7. `burn(uint256)`
   Burns an NFT.

8. `transferFrom(address from, address to, uint256 tokenId)`
   Transfer the NFT with the given token ID from the provided from address to the desired recipient.

9. `safeTransferFrom(address from, address to, uint256 tokenId, bytes data)`
   Similar to `transferFrom` but with additional guarantees about the recipient being compatible with the NFT.

Properties

1. Transfer functions only changes the ownership of the token ID to be transferred and no other token ID ✓
2. Consistency of `tokenOfOwnerByIndex` - unless the holder transfers their NFT, and the token is having the same index for the owner, then it cannot be assigned a different owner ✓

Verification of Zesty Vault

Store of Zesty NFTs. The vault is abstract and is used by both versions of Zesty Market.

Functions

1. `authorizeOperator(address _operator)`

2. `revokeOperator(address _operator)`

3. `depositZestyNFT(uint256)` (internal)

4. `withdrawZestyNFT(uint256)` (internal)

5. `getDepositor(uint256 tokenID): address`
   Getter for NFT deposits, mapping a token ID to its depositor.

6. `getOperator(address depositor): address`
   Getter for NFT deposit operators, mapping a depositor to its operator.

7. `getZestyNFTAddress(): address`

Return the address of the underlying NFT.

Properties

1. Cannot transfer NFTs that do not belong to the sender 🔁

Zesty Market V1.1

Functions

Some of the functions are allowing for batching several operations of the same type, by treating each of the arguments as an array, and enforcing all arguments arrays have the same length. We mark such functions below, but present them as if they were a single operation.

1. `buyerCampaignCreate(string uri)`
   Creates a new campaign identified by the given URI.

2. `sellerNFTDeposit(uint256 tokenId, uint8 autoApprove)`
   A seller deposits an NFT representing an ad space, and whether bids will be autoapproved or not.

3. `sellerNFTWithdraw(uint256 tokenId)`
   A seller withdraws an NFT representing an ad space, assuming all auctions have been closed.

4. `sellerNFTUpdate(uint256 tokenId, uint8 autoApprove)`
   A seller updates auto approval for the NFT with the given token ID.

5. `sellerBan(address a)`
   A seller bans a potential buyer with the given address a.

6. `sellerUnban(address a)`
   A seller unbans a potential buyer with the given address a.

7. `sellerAuctionCreate(uint256 tokenId, uint256 auctionTimeStart, uint256 auctionTimeEnd, uint256 contractTimeStart, uint256 contractTimeEnd, uint256 priceStart)`
   A seller creates a single auction. (In code: Batched version.)

8. `sellerAuctionCancel(uint256 sellerAuctionId)`
   A seller cancels a single auction (assuming no bids). (In code: Batched version.)

9. `sellerAuctionBid(uint256 sellerAuctionId, uint256 buyerCampaignId)`
   A buyer bids in a single auction. (In code: Batched version.)

10. `sellerAuctionBidCancel(uint256 sellerAuctionId)`
    A buyer cancels a bid in a single auction. (In code: Batched version.)

11. `sellerAuctionApprove(uint256 sellerAuctionId)`
    A seller approves a buyer's bid. (In code: Batched version.)

12. `sellerAuctionReject(uint256 sellerAuctionId)`
    A seller rejects a buyer's bid, returning bid sum to buyer. (In code: Batched version.)

13. `contractWithdraw(uint256 contractId)`
    A seller withdraws a fulfilled contract, receiving bid sum. (In code: Batched version.)

14. `getSellerNFTSetting(uint256 tokenId): (uint256 tokenId, address seller, uint8 autoApprove, uint256 inProgressCount)`
    Getter for the NFT settings (auto approval flag) of the given token ID.

15. `getSellerAuctionPrice(uint256 sellerAuctionId): uint256`
    Computes the relative price taking into account the elapsed time.
    (Math: decreases monotonically until contract time end, beginning from the start price.)

16. `getSellerAuction(uint256 id): (address seller, uint256 tokenId, uint256 auctionTimeStart, uint256 auctionTimeEnd, uint256 contractTimeStart, uint256 contractTimeEnd, uint256 priceStart, uint256 pricePending, uint256 priceEnd, uint256 buyerCampaign, uint8 buyerCampaignApproved)`
    Getter for the seller auction for the given ID.

17. `getBuyerCampaign(uint256 id): (address buyer, string uri)`
    Getter for the buyer campaign for the given ID.

18. `getContract(uint256 id): (uint256 sellerAuctionId, uint256 buyerCampaignId, uint256 contractTimeStart, uint256 contractTimeEnd, uint256 contractValue, uint8 withdrawn)`
    Getter for the contract with the given ID.

Properties

1. For a token that is mapped to a seller, auto approve can be either 1 (“FALSE”) or 2 (“TRUE”), and if the token is not mapped to a seller, then auto approve is 0. ✓

2. Seller auction price rules:
   a. Price decreases monotonically in time ✓
   b. Price may only decrease in transitions ✓
   c. Price at auction time start is equal to the specified start price ✓

3. Batchable operations are additive: we get the same erc20 effect for the sender and the market contract for each batchable operation (e.g. bid) when running on [a1, a2] and on ([a1], [a2]).
   a. Additivity of bid ✓
   b. Additivity of approve ✓
   c. Additivity of bid cancel ✓
   d. Additivity of reject ✓
   e. Additivity of withdraw ✓

4. Either `priceEnd` or `pricePending` should be zero. Both cannot be non-zero. ✓

5. Before bidding and setting the winning campaign, there cannot be a `priceEnd` or `pricePending` setting ✓

6. Once `priceEnd` is set, `pricePending` must be zero and the campaign must be approved ✓

7. Auctions above the seller auction count must be uninitialized ✓

8. Campaigns above the buyer campaign count must be uninitialized, and campaign ID 0 is never initialized ✓

9. Contracts above the contract count must be uninitialized ✓

10. NFT depositor is the same as the seller set in NFT settings ✔️

11. An auction's token ID must have the same seller in both auctions struct and settings struct ✔️

12. Once a contract is marked as withdrawn this cannot be changed ✔️

13. Solvency. The token balance of the market must be greater than the sum of all pending auction bids, plus all ended auction bids, minus the sum of all withdrawn contract valus. ✔️

14. Monotonicity of counters
    a. Buyer campaign count should monotincally increase and start at 1 ✔️
    b. Seller auction count should monotonically increase and start at 1 ✔️

15. Time ranges validity: For a created auction, auction time end must be greater than auction time start, contract time end must be greater than contract time start and auction time end ❌

16. All calls that have potential external calls protect against reentrancy ✔️

17. When an NFT is deposited in the contract, owner shouldn't be able to withdraw it unless the in progress count is set to 0 🔁

18. An auction can have a price if and only if it has a buyer campaign ✔️

19. If an NFT has a seller, then it must be owned by the contract 🔁

20. The variables for `txToken` and `txTokenAddress` hold the same value. ✔️

21. The difference in price pending + price delta is equal to the difference in the balance of the buyer and/or market, where every batchable operation is assumed to take just one element. ✔️

```
let price(auctionId) =
    getAuctionPricePending(auctionId) + getAuctionPriceEnd(auctionId)
let delta_price(auctionId) = p' - p
    in { p = price(auctionId) } op(); { p' = price(auctionId) }
let delta_balance(u) = b' - b
    in { b = token.balanceOf(u) } op(); { b' = token.balanceOf(u) }

{ buyerBalance = token.balanceOf(buyer)
  ∧ buyer ≠ market
  ∧ hasBuyer = getAuctionBuyerCampaign(auctionId) ≠ 0
  ∧ buyer = getBuyer(getAuctionBuyerCampaign(auctionId)) }
    op() where op is sellerAuctionApproveBatch([auctionId])
        ∨ op is sellerAuctionBidCancelBatch([auctionId])
        ∨ op is sellerAuctionRejectBatch([auctionId])
{ hasBuyer ⇒ delta_price(auctionId) = delta_balance(buyer)
  ∧ delta_price(auctionId) = delta_balance(market) }


{ buyerBalance = token.balanceOf(buyer)
  ∧ buyer ≠ market
  ∧ buyer = getBuyer(c) }
    op() where op is sellerAuctionBidBatch([auctionId], c)
{ delta_price(auctionId) = delta_balance(buyer)
  ∧ delta_price(auctionId) = delta_balance(market) }
```

22. Revert conditions for bid cancellation ✔

23. Revert conditions for auction cancellation ✔❌

Zesty Market V2

The Zesty Market V2 is very similar in structure and functionality to V1, but includes better checking of actual contract fulfillment by the seller.

Most of the rules for V1 can be applied to V2 as well, and no new violations were found.