CD SECURITY

AUDIT REPORT

Dragon Stone

February 2024

Introduction

A time-boxed security review of the **Dragon Stone** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **DragonStone**

Dragon Stone is an NFT collection of 20 000 NFTs that utilizes the `EC721A` standard. A special English Auction will be held for a specific NFT. Then, only whitelisted accounts will be able to mint the NFTs. There isa public `mint` function that is protected by a pausing mechanism and can be used in the future for public sale.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 246495cbc03f993f4eb8d25ebf3421ee4086d67b**

Scope

The following smart contracts were in scope of the audit:

- `EnglishAuction.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 2 issues
- Medium: 2 issues
- Low: 1 issues
- Informational: 5 issues

Findings Summary

ID	Title	Severity
[C-01]	The auction can be bricked	Critical
[H-01]	The `bid` function will revert because of a wrong check	High
[M-01]	Insufficient input validation	Medium
[M-02]	Owners can rug the users and not send anything	Medium
[L-01]	Use two-step ownership transfer approach	Low
[I-01]	Missing event emissions in state changing methods	Informational
[I-02]	Prefer Solidity Custom Errors over `require` statements	Informational
[I-03]	Change function visibility `public` to `external`	Informational
[I-04]	Use newer and stable pragma statement	Informational
[I-05]	Redundant code	Informational

Detailed Findings

[C-01] The auction can be bricked

Severity

Impact: High because the auction cannot be ended and no more bids will be accepted

Likelihood: High as it is easy to do it through a flash-loan or even unintentionally

Description

The English-style auction allows users to bid for a specific NFT and the user with the highest bid will win it. However, every user can decide to withdraw their bid by calling `withdrawBid` and get their money back:

```
    function withdrawBid() public auctionActive nonReentrant {
        uint256 bidAmount = auction.bids[msg.sender];
        require(bidAmount > 0, "No bid to withdraw");
        auction.bids[msg.sender] = 0;
        (bool os, ) = payable(msg.sender).call{value: bidAmount}("");
        require(os, "Transfer to bidder failed");
        emit BidRefunded(msg.sender, bidAmount);
    }
```

This means that the highest bidder can decide to withdraw their money at any time. This opens an attack vector that can easily brick the bidding functionality because the `auction.highestBid` is not reset.

For example, a user can take a flash loan (let's say 1000 ETH for simplicity), bid with the whole amount, and then withdraw. This will update the `auction.highestBid` to 1000 ETH but the funds will not be in the contract. Thus, the auction can't be ended as the `endAuction` will revert because of insufficient funds.

```
    function endAuction() public onlyOwner auctionActive {
        auction.active = false;
        if (auction.highestBidder != address(0)) {
            (bool os, ) = payable(owner()).call{value: auction.highestBid}
("");
            require(os, "Transfer to owner failed");
            emit AuctionEnded(auction.highestBidder, auction.highestBid);
        }
    }
```

This can happen unintentionally if the highest bidder needs to withdraw their ETH for some reason and no one else is willing to bid more.

Recommendations

There is no easy fix to this issue. An example solution is to check if the `withdrawBid` is the highest bid and reset it but it will need to inform all of the other users.

[H-01] The `bid` function will revert because of a wrong check

Severity

Impact: High as the `bid` function will revert wrongly

Likelihood: Medium because it will happen when users want to bid more than once

Description

The `bid` function can be called when users want to bid again and increase their bid. This happens by adding the current `msg.value` to all the previous bids and calculating it in `newBidTotal` variable:

```
    function bid() public payable auctionActive {
        require(msg.value > auction.highestBid, "Bid not high enough");
        require(msg.value >= minimumBid, "Bid below minimum bid");

        // Allow previous bids to be overridden
        uint256 newBidTotal = auction.bids[msg.sender] + msg.value;
        require(
            newBidTotal > auction.highestBid,
            "Total bid not high enough to become highest bidder"
        );
```

The problem is that the call can revert because of the first `require` statement that checks if the current `msg.value > auction.highestBid`. This could lead to the following scenario:

1. User A bids 3 ETH.
2. after that, User B bids 3.5 ETH, and now `auction.highestBid` = 3.5 ETH.
3. User A wants to bid again and sends 1 ETH.
4. The call reverts because of the first check - 1 ETH < 3.5 ETH.
5. However, the `newBidTotal` will be 4 eth which will be higher than `auction.highestBid`.

Thus, the function call will revert wrongly in step 4.

Recommendations

Consider removing the first check and moving the second check after `newBidTotal` is calculated.

```diff
-        require(msg.value > auction.highestBid, "Bid not high enough");
-        require(msg.value >= minimumBid, "Bid below minimum bid");

        // Allow previous bids to be overridden
        uint256 newBidTotal = auction.bids[msg.sender] + msg.value;
+        require(msg.value >= minimumBid, "Bid below minimum bid");
        require(
            newBidTotal > auction.highestBid,
            "Total bid not high enough to become highest bidder"
        );
```

[M-01] Insufficient input validation

Severity

Impact: High, because some of these functions are setting important values which can be problematic

Likelihood: Low, as it requires a malicious/compromised owner account or an owner input error

Description

Throughout the codebase, there are couple of functions where important input validation is missing. The inputs are not constrained at all.

- `setMinimumBid()`
- `setMaxSupply()`
- `setCost()`
- `setPhase()`

Recommendations

You can create a check where `exampleParam` should be less than x or greater than y, otherwise revert the transaction.

[M-02] Owners can rug the users and not send anything

Severity

Impact: High, because the users will be left with nothing and their funds will be stolen

Likelihood: Low, because it requires a malicious owner.

Description

The auction is designed to reward the winning bid with a specific NFT. However, when the auction ended, nothing was sent to the winner, only the funds were transferred to the owner:

```
    function endAuction() public onlyOwner auctionActive {
        auction.active = false;
        if (auction.highestBidder != address(0)) {
            (bool os, ) = payable(owner()).call{value: auction.highestBid}("");
            require(os, "Transfer to owner failed");
            emit AuctionEnded(auction.highestBidder, auction.highestBid);
        }
    }
```

This opens up a rug factor as the owners will withdraw `auction.highestBid` but can decide to not send the NFT to the winner of the auction.

Recommendations

Transfer the NFT inside the `endAuction` call so the users are sure that there is a guaranteed prize.

[L-01] Use two-step ownership transfer approach

The `owner` role is crucial for the protocol as there are a lot of functions with the `onlyOwner` modifier. Make sure to use a two-step ownership transfer approach by using `Ownable2Step` from OpenZeppelin as opposed to `Ownable` as it gives you the security of not unintentionally sending the `owner` role to an address you do not control.

[I-01] Missing event emissions in state changing methods

It's a best practice to emit events on every state changing method for off-chain monitoring. The setter functions and the `withdraw` method are missing event emissions, which should be added.

[I-02] Prefer Solidity Custom Errors over `require` statements

Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well. Remove all `require` statements and use Custom Errors instead.

[I-03] Change function visibility `public` to `external`

If the function is not called internally, it is cheaper to set your function visibility to external instead of public. Make all `public` functions `external`.

[I-04] Use newer Solidity version with a stable pragma statement

Using a floating pragma `>=0.8.9 <0.9.0` statement is discouraged as code can compile to different bytecodes with different compiler versions. Use a stable pragma statement to get a deterministic bytecode. Consider using a stable 0.8.19 version to make sure it is up to date.

[I-05] Redundant code

The below variables are used for tracking numbers but are not used anywhere:

```
    uint256 public phase = 0;
    mapping(address => uint256) public numberOfBids;
```

Consider removing them and tracking them off-chain.