zkAuction Audit Report

Version 1.0

veridise.com

November 5, 2023

zkAuction Audit Report

Upal Chowdhury

Nov 5, 2023

Prepared by: Upal Chowdhury

Table of Contents

Protocol Summary

This protocol implements a Sealed-Bid auction. The auction starts when an admin (or rather the owner of the auctioned token) creates an auction and invites users to participate. For a user to bid in the auction, they must first deposit funds into the auction which effectively represents their maximum bid in the auction. All funds in the protocol are represented as cryptographic notes that hide the exact value of the note, but can be spent via the ZK circuits to hide bidding values. Note that due to the nature of the deposits, the total value that a user may spend is technically public, but we believe that is an acceptable trade-off. During an auction, a user may then use any amount of their deposited funds

to submit a bid via the protocol's circuits. Once they do so, these funds are locked in the auction and may not be retrieved or withdrawn until after the auction has been completed. After a certain amount of time, bidding will close and the auction will enter the reveal phase. During this phase, users have a certain amount of time to reveal their bids where the highest revealed bid wins. Note, users do not have to reveal their bid which allows some users to bluff others (by using their public maximum) but that only increases the fun of the auction. After the reveal period ends, the winner auction can distribute the funds from the winning bid to the admin/beneficiary and the NFT to the winner. All other users at this point may recover their bids, which places the funds back in the user's liquid balance (i.e. at this point it can be withdrawn).

Disclaimer

We make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

- Critical: the project will definitely get hacked upon deployment.
- High: the project will likely get hacked, but an attacker will need some luck on their side.
- Medium: the project will likely get hacked, but the impact of the hack won't be detrimental.
- Low: the project might get hacked, but the attack surface and impact are minimal.
- Warning: the project won't get hacked, but code quality can be improved.
- Info: the project won't get hacked, but the developers follow bad engineering practices.

Audit Details

Prepare a audit report including the issues in the code for both the circuits and the contracts.

Executive Summary

Overall it was well constructed protocol. Although no major issues found, some areas that can be improved upon such as adding reentrancy guard while making external call, proper authentication,

detail error messaging and gas optimizations. Due to time limitation no all part of the circuits are tested but note that the circuit's security relies on the assumption that the prover cannot generate a valid proof for an incorrect index without knowing the corresponding siblings that hash to the root. This is fundamental to the zero-knowledge property of the proof: the prover can demonstrate knowledge of a valid index and siblings without revealing what they are.

Issues found

Circuits

Bid.cirom

——————— # Medium 1. balcheck component output is not constrained to any signal which could cause issues such as between balance should be greater than bid amount. component balCheck = LessEqThan(252); balCheck.in[0] <== bid; balCheck.in[1] <== balance;

Low

2. Invalid proof can be generated and verified by changing auctionRoots. It can be reproduced using steps here https://github.com/upalchowdhury/veridiseWorkshop/blob/main/circuits/Verifying%20invalid%20pr
   This will need to be further investigated to ensure if this is an issue thus marking it as low for now.

Set.circom

———————-

Medium

Zero-Product Vulnerability: In SetMembership, if element is equal to any element in set, the product will be zero. However, if an element in set is repeated, it could also produce a zero product even if the element is not in the set. This could be a logical flaw if duplicates are allowed in the set without proper handling. Ensure that there are no duplicate values in set. duplicates could cause the zero-product issue we discussed earlier.If the logic expects no duplicates in the set, then add constraints to check that all elements in set are unique.

Tree.circom

———————

Medium

Underconstrained: If s could be any value other than 0 or 1, the circuit would not function as intended. Therefore, it would be better to enforce s to be binary if it's not guaranteed by the external context.The enforcement of s being binary is crucial because it determines which of the two inputs (hashes[i] or siblings[i]) will be selected. The line indices[i] * (1 - indices[i]) === 0; enforces that indices[i] is either 0 or 1, which is expected for the selector s in the Decider.

Membership.circom

————————

Low

Unconstrained The receiver signal is not constrained. Given that this is going to be an ethereum address might not be critical but It should be constrained to ethereal address format of 20 bytes

Contracts

AuctionEscrow:

————————— # Gas 1.Gas Optimization In the deposit and withdraw functions, there are multiple calls to _balanceHash. Depending on the gas costs of the Poseidon hash function, it might be more efficient to calculate the hash once and store it in a local variable, rather than calling the function multiple times with the same parameters.

Low

2. Possible underflow or error as no check on amount to be less than curBal function withdraw(uint256 nonce, uint256 curBal, uint256 amount, address receiver, uint256 commitment, uint256 nullifier, uint256[] calldata escrowSiblings, uint256[8] calldata proof) external {

require(commitmentInd[commitment] != 0); require(!usedNullifiers[nullifier]); usedNullifiers[nullifier] = true;

```
membershipVerifier.verifyProof(nonce, escrowTree.root, curBal,
    receiver, nullifier, commitment, proof);

uint256 oldLeaf = _balanceHash(commitment, curBal);
uint256 newLeaf = _balanceHash(commitment, curBal - amount);
```

The calculation curBal - amount in withdraw should be safe, but if amount were ever larger than curBal, this would revert due to underflow.

Medium

3.Re-Entrancy on deposit The deposit function calls an internal function _depositToExisting which updates the state, and then it proceeds to transfer tokens from the sender to the contract. If the ERC20 token used is implemented in a way that allows for reentrancy, there could be a reentrancy attack here.

function deposit(uint256 commitment, uint256 curBal, uint256 amount, uint256[] calldata escrowSiblings) external { if(commitmentInd[commitment] == 0) { deposit(commitment, amount); return; }

```
_depositToExisting(commitment, curBal, amount, escrowSiblings);
require(token.transferFrom(msg.sender, address(this), amount));
emit Deposit(commitment, curBal, amount, commitmentInd[commitment]);
```

}

The potential issue here is that the state is updated before the token transfer. If the ERC20 token does not adhere to the standard and implements a re-entrant transferFrom, it could call back into the contract and lead to unexpected behavior.

4.Initial Deposit Check The initial deposit check uses the following condition: solidity

`require(commitmentInd[commitment] == 0);` This check assumes that commitmentInd for a given commitment starts at 0 and that a 0 value indicates no previous deposit. This is the default behavior for uninitialized mapping values in Solidity. However, the code should ensure that a 0 index is indeed invalid for a legitimate commitment. If 0 is a valid index, then the contract would incorrectly prevent deposits for that index.

Info

4. Event Information The Deposit and Withdraw events log the commitment and the amount, but do not log the sender or receiver of the transaction. Including the sender and receiver in the events could be helpful for off-chain tracking and auditing of the contract's interactions.

MembershipVerifer.sol

———————————

Info

Lack of Upgradability and No error message like revert is noticed.

Auction.sol

———————

High

1.Frontrunning : the reveal function is a potential target for front-running attacks. A malicious actor watches the mempool for reveal transactions.They identify a reveal transaction that includes a high bid amount. Before the original transaction is mined, the attacker submits their own reveal transaction with an even higher bid amount and a higher gas fee to ensure that it gets mined first. Recommendation is to use a commit-reveal scheme with a randomization factor that is not known until after the reveal period, making it hard to determine the winning bid in advance. And implement a time-lock on bids to prevent new bids from being revealed immediately after another reveal.

Info

2. Error Messages Many require statements lack error messages. Providing specific error messages can help with debugging and understanding why a transaction failed.

High

3.Reentrancy Concerns: The nft.safeTransferFrom and token.transfer are external calls that could potentially be exploited for reentrancy. It is good practice to perform state changes before making these calls. The state change here (auctions[auctionId].winningAmt = 0) is done after safeTransferFrom but before token.transfer. This should be rearranged to precede all external calls. Reentrancy guard should be used whenever calling external contracts.

Medium

4.No Check on Winner or Amount: There is no check to ensure that auctions[auctionId].winner and auctions[auctionId].winningAmt are set to non-zero values. This check is important to ensure that the function does not execute with default values (in case something went wrong during the auction).

Info

5.Error Handling: The require statement is used to ensure the transfer is successful. However, if the transfer fails, it will revert the entire transaction, including the NFT transfer. Depending on the desired behavior, this might be too harsh, and a check-effects-interactions pattern might be more appropriate.

Medium

6.Authentication: There is no authentication on who can call the refund and distribute function. This implies that anyone can attempt to refund any bid after the auction is over. There should be checks to ensure that only the bidder or a legitimate party can call this function to refund a bid.

Info

7. Contract Upgradeability There is no apparent mechanism for upgrading the contract. If a bug is found or an improvement is needed, there would be no way to upgrade the contract without migrating to a new contract and updating all references to it.

Gas

8. Gas Limit and Loops The addMembers function uses a loop to add multiple members, which could potentially run out of gas if given a large array of identityCommitments. It has an unchecked increment in the loop, which is good for gas optimization, but be aware of the risks of running out of gas if the array is too large.

Info

9. Timestamp Dependence The contract uses block.timestamp for auction timing. While generally safe for longer periods (like days or weeks), it is important to remember that In PoS, a validator is chosen to propose a block, and they are responsible for choosing the block timestamp. The Ethereum protocol requires that the timestamp of a block must be greater than the timestamp of the previous block, but there is still a small degree of flexibility that allows validators to manipulate timestamps to some extent.