zkAuction

Smart Contract Security Assessment

November 5, 2023

Prepared for:

Veridise

Prepared by:

Andrea De Dominicis

Contents

1 Executive Summary

I conducted a security assessment for zkAuction from November 4th to November 5th, 2023. During this engagement, I reviewed zkAuction code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

The goal of the audit is to assess project code (with any associated specification, documentation) and alert of potential security-related issues that need to be addressed to improve security posture, decrease attack surface and mitigate risk.

1.2 Non-goals and Limitations

Audit **is not** a security guarantee of “bug-free” code by any stretch of imagination but a best-effort endeavor by trained security experts operating within reasonable constraints of time, understanding, expertise and of course, decidability.

Due to time constraints, I was unable to fully exploit or thoroughly test some of the vulnerabilities, which means that some findings may result in false positives.

1.3 Risk Score

Using the following severity levels:

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.

1.4 Results

During the assessment on the scoped zkAuction contracts, **10 findings** have been discovered. Two critical issues were found. Three were of medium impact, one was of low impact and four were of Info impact.

2 Introduction

2.1 About zkAuction

The project 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).

2.2 Scope

The engagement involved a review of the following targets:

Smart Contracts	Circuits
Auction.sol *base/AuctionAdmin.sol* *base/BidVerifier.sol* *base/BidVerifier/verifier.sol* *base/MembershipVerifier/verifier.sol* *base/AuctionEscrow.sol* *base/MembershipVerifier.sol* *base/IncrementalBinaryTree.sol* *interfaces/IMembershipVerifier.sol* *interfaces/IAuctionAdmin.sol* *interfaces/IBidVerifier.sol* *interfaces/IAuction.sol*	*identity.circom* *bid.circom* *set.circom* *tree.circom* *membership.circom* *mains/balance_membership.circom* *mains/auction_bid.circom*

2.3 Project Timeline

The key dates of the engagement are detailed below.

November 3, 2023	Audit Challenge Release
November 3, 2023	Audit Challenge Description
November 4, 2023	Audit
November 5, 2023	Report

3 Detailed Findings

3.1 [Critical] Unconstrained subcomponent output circuit

Description

Output from subcomponent call is not used, if any outputs are unused or used but not referenced in any of the containing component's constraints.

```
59: component balCheck = LessEqThan(252);
60: balCheck.in[0] <== bid;
61: balCheck.in[1] <== balance;
```

bids.circom#59-61

Impact

A malicious actor could exploit these missing constraints to create valid proofs for unintended statements and incur serious consequences.

***Note**: Picus cannot determine whether the circuit is properly constrained, did not have time to properly test this.*

Remediation

Properly constrain output signal from subcomponent.

3.2 [Critical] Non-Deterministic Witness for circuit

Description

The Non-Deterministic Witness issue occurs when dataflow is dependent on conditional branches or conditional assignments, so the witness is non-deterministic.

```
    for (var i = 0; i < nLevels; i++) {
        hashers[i] = Poseidon(2);
        deciders[i] = Decider();
        indices[i] <-- (index & (1 << i) == 0) ? 0 : 1; // NDW
        indices[i] * (1 - indices[i]) === 0;

        deciders[i].in[0] <== hashes[i];
        deciders[i].in[1] <== siblings[i];
        deciders[i].s <== indices[i];

        hashers[i].inputs[0] <== deciders[i].out[0];
        hashers[i].inputs[1] <== deciders[i].out[1];

        hashes[i + 1] <== hashers[i].out;
    }
```

tree.circom#28-45

Impact

Non-deterministic witnesses in zk-circuits can introduce a significant risk to the security and functionality of the zero-knowledge proof system, as unconstrained values could allow for the construction of bogus proofs.

Remediation

To mitigate these risks, it's crucial to design zk-circuits to be as deterministic as possible. This involves careful consideration of the constraints and input data to ensure that the proof generation process is consistent and secure.

3.3 [Critical] Unconstrained input circuit *(may be FP)*

Description

Input signal is not used in any constraint.

```
    template Membership(nLevels, historySize) {
        signal input identityNullifier;
        signal input identityTrapdoor;
        signal input index;
        signal input treeSiblings[nLevels];

        signal input nonce;
        signal input roots[historySize];
        signal input leafData;

        signal input receiver; // not constrained

        signal output nullifierHash;
        signal output identityCommitment;
```

membership.circom#48

Impact

If the circuit is underconstrained, a malicious actor may be able to create new valid proofs by taking an existing proof and simply changing a public input that is unconstrained.

***Note**: after simplifying the circuit and running Picus on it, Picus confirms that the circuit is not underconstrained, considering that when Picus output confirms the proper constrainment it should be correct. Did not have time to properly test this.*

Remediation

Properly constrain input signals.

3.4 [Medium] Reentrancy

Description

A state variable is changed after a contract uses call.value. The attacker uses a fallback function—which is automatically executed after Ether is transferred from the targeted contract—to execute the vulnerable function again, before the state variable is changed.

```
function distribute(uint256 auctionId) afterAuction(auctionId)
afterAuction(auctionId) external {
     nft.safeTransferFrom(address(this), auctions[auctionId].winner,
     auctions[auctionId].tokenId); // External call
     uint256 amt = auctions[auctionId].winningAmt; // State written after
     call
     auctions[auctionId].winningAmt = 0;
     require(token.transfer(auctions[auctionId].admin, amt));
     emit Distribute(auctionId, auctions[auctionId].tokenId,
     auctions[auctionId].winner, amt);
}
```

Auction.sol#202-208

Impact

The attacker can repeatedly transfer money to the auction winner until execution stops.

***Note**: I wanted to test reentrancy as well but unfortunately didn't have enough time to do proper exploitation. I could have done it through hardhat testing script, but I'm more familiar with Foundry, so considering time constraints I could not.*

Remediation

Apply the check-effects-interactions pattern

3.5 [Medium] Unused return

Description

The return value of an external call is not stored in a local or state variable.

```
    function deposit(uint256 commitment, uint256 amount) public {
        require(commitmentInd[commitment] == 0);
        commitmentInd[commitment] = escrowTree.numberOfLeaves;
        uint256 bal = _balanceHash(commitment, amount);
        escrowTree.insert(bal); // ignored return value
        require(token.transferFrom(msg.sender, address(this),
        amount));
        emit Deposit(commitment, 0, amount,
        commitmentInd[commitment]);
    }
```

AuctionEscrow.sol#32-39

```
    function _addMember(uint256 auctionId, uint256 identityCommitment)
    internal virtual {
        if (getMerkleTreeDepth(auctionId) == 0) {
        revert AuctionDoesNotExist();
        }

        uint256 leaf = PoseidonT3.hash([identityCommitment,
        auctionId]);
        merkleTrees[auctionId].insert(leaf); // ignored return value

        uint256 merkleTreeRoot = getMerkleTreeRoot(auctionId);
        uint256 index = getNumberOfMerkleTreeLeaves(auctionId) - 1;

        TreeHistory storage history = histories[auctionId];
        history.front = (history.front + 1) % 20;
        history.historicRoots[history.front] = merkleTreeRoot;

        emit MemberAdded(auctionId, index, identityCommitment,
        merkleTreeRoot);
    }
```

AuctionAdmin.sol#36-52

```
    bidVerifier.verifyProof(nonce, history, escrowTree.root,
    balIndex, nullifierHash, bidLeaf, newBalLeaf, proof);
    _setBal(balIndex, balLeaf, newBalLeaf, balSiblings);

    uint256 ind = auctions[auctionId].bids.numberOfLeaves;
    auctions[auctionId].bids.insert(bidLeaf); // ignored return
    value
    emit Bid(auctionId, nullifierHash, ind, bidLeaf);


Auction.sol#168-174
```

Impact

Failure to properly handle these return values or understand the nuances between these methods can lead to vulnerabilities in the smart contract, opening up opportunities for exploits.

Remediation

- Use Transfer Over Send: whenever possible, use transfer() instead of send() since the former reverts the transaction if the external call fails.
- Check Return Values: Always validate the return value of send() or call() functions to take appropriate actions if they return false.

3.6 [Low] Events Reentrancy

Description

An event is emitted after a contract uses call.value. The attacker uses a fallback function—which is automatically executed after Ether is transferred from the targeted contract—to execute the vulnerable function again, before the event is emitted.

```
/// @dev See {ISemaphore-createGroup}.
function createAuction(
    uint256 auctionId,
    address admin,
    uint256 duration,
    uint256 tokenId
    ) external override {
    _createAuction(auctionId);

    auctions[auctionId].admin = admin;
    auctions[auctionId].auctionStart = block.timestamp + 1 days;
    auctions[auctionId].auctionDuration = duration;
    auctions[auctionId].tokenId = tokenId;
    auctions[auctionId].bids.init(20, 0);

    // external call
    nft.safeTransferFrom(msg.sender, address(this), tokenId);

    // event emit
    emit AuctionCreated(auctionId, tokenId, admin, duration);
}
```

Auction.sol#52-70

Impact

If the external call nft.safeTransferFrom re-enters AuctionCreated, the event generated could not correspond to truth as the attacker has executed the transfer many times, but only one event is emitted. This may cause issues for off-chain components that rely on the values of events e.g. checking for the amount deposited to an auction..

Remediation

Apply the check-effects-interactions pattern

3.7 [Info] State variables could be declared immutable

Description

State variables that are not updated following deployment should be declared immutable to save gas.

```
IBidVerifier public bidVerifier;
IERC721 nft;
```

Auction.sol#13-14

```
IERC20 token;
IMembershipVerifier membershipVerifier;
```

AuctionEscrow.sol#16-17

Impact

It does not really have a security impact or specific risks associated, but it is bad for optimization and gas costs.

Remediation

Add the immutable attribute to state variables that never change or are set only in the constructor.

3.8 [Info] Unused state variable

Description

Unused state variable.

```
library IncrementalBinaryTree {
    uint8 internal constant MAX_DEPTH = 32;
    uint256 internal constant SNARK_SCALAR_FIELD =
    21888242871839275222246405745257275088548364400416034343698204186575
    808495617;


IncrementalBinaryTree.sol#21-24
```

Impact

It does not really have a security impact or specific risks associated, but it is bad coding practice.

Remediation

Remove unused state variables.

3.9 [Info] Use of Assembly

Description

The use of assembly is error-prone and should be avoided.

Impact

Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.

Remediation

This is flagged as 'Info.' While it is true that using assembly is generally considered risky and should be avoided unless necessary, it is a requirement in this specific project due to the zk-circuits and other low-level components, which cannot be achieved within the standard Solidity context.

3.10 [Info] Different versions of Solidity are used

Description

Multiple solidity versions are used.

Impact

Using different solidity versions across the project could cause unexpected behaviors.

Remediation

Use a single version of Solidity where possible.