zkAuction Bugs

This document describes the bugs we introduced in the code for zkAuction, the codebase for Day 5 of the workshop. The codebase was composed of two modules; a set of smart contracts written in Solidity and a set of ZK circuits written in Circom. We introduced 3 types of issues; smart contract issues, ZK issues, and protocol issues. “Smart contract issues” and “ZK issues” describe issues in the smart contracts and ZK circuits (respectively), whereas “protocol issues” describes issues in any part of the codebase related to the interaction between users and the system.

Issues in the ZK Circom circuits

1. Zero root always passes check

This bug is in the `SetMembership` template in `set.circom`. This template checks if an element belongs to a set. It does so by computing the product of the differences between each element and the input. If the end product is 0 then the element is in the set.

As an example let’s check if 7 is part of $\{1, 4, 8, 7, 3\}$ ($p = 13$). The circuit will produce a computation similar to $7 \times (1 - 7) \times (4 - 7) \times (8 - 7) \times (7 - 7) \times (3 - 7) = 7 \times 7 \times 10 \times 1 \times 0 \times 9 = 0$. The witness generator will only generate a valid proof if the final value is zero (line 17 of the circuit). If we try with a value that is not in the set like 5 we will get 7.

Since the element is used to start the multiplication if the element we want to check is 0 then the resulting multiplication will always be 0 which means that regardless of 0 belonging to the set in question or not the witness generator will always produce a valid proof.

2. Underconstrained index used when computing merkle tree root

This bug is in the `MerkleTreeInclusionProof` template in `tree.circom`. The `index` input signal is not used in any of the constraints declared in the template which means that it

is underconstrained and could lead to the forgery of proofs.

3. Nondeterministic witness calculation

This issue is in the `MerkleTreeInclusionProof` template in `tree.circom`. The `index` input signal is used in the conditional expression of a ternary expression. This is considered a code smell and is not recommended. While not a bug it could facilitate the apparition of others (see issue #2 above).

4. Unconstrained public input

This bug is in the `Membership` template in `membership.circom`. The `receiver` public input signal is not constrained. Since it's a public input signal an attacker could fake proofs with different values that a proof validator will accept as long as the other values are the same.

5. Missing range check on the inputs

This bug is in the `Bid` template in `bid.circom`. Input signals `bid` and `balance` are used in a `LessEqThan` component. This component requires its inputs to be of a certain bit length. In this particular case, of 252 bits. This is part of the assumptions for `LessEqThan` to function as expected and is not checked by the template, as the bit length check is deferred to the user. However the bit length is not checked for these two signals, which means that their maximum bit length is 254, larger than what the component expects.

6. Missing constraint on LessEqThan output signal

This bug is in the `Bid` template in `bid.circom`. The output of `LessEqThan` is not used by the caller. This means that the witness generator is not actually checking if $bid \leq balance$. This allows an attacker to bid higher than expected.

7. Arithmetic overflow

As a consequence of the previous issue, in line 69 of `bid.circom`, if $bid > balance$ an arithmetic underflow will happen. Since Circom uses finite fields for arithmetic this allows the attacker to bid a really high number because the negative number will wrap around p.

Issues in the smart contracts

1. Entries in merkle tree are not checked

Entries in the merkle tree implemented in `IncrementalBinaryTree.sol` are not checked for size. These entries must fit within the field in order to guarantee consistency. If a value is larger than p the circom runtime will wrap around it, producing unexpected results.

2. Hard coded value provides permanent member

The initialization of the merkle trees produces a hard coded leaf with the value of 0. This means that ID 0 is a member of all trees, whether that ID is actually a member of the auction or not.

This initialization happens in:

- `AuctionEscrow.sol` line 24
- `AuctionAdmin.sol` line 25
- `Auction.sol` line 69

3. User at index 0 of EscrowTree has funds locked

In `AuctionEscrow.sol` the methods `deposit` in line 32, `deposit` in line 42 and, `withdraw` in line 52 implement checks over `commitmentInd`. The checks in the `deposit` methods allow the commitment with index 0 to deposit funds. However, the check in `withdraw` forbids the commitment with index 0 to complete the transaction. This means that the funds inserted by the user with that index are locked and cannot be withdrawn after the auction.

4. Root membership check allows non-approved users to participate with forged merkle tree root

The check of membership implemented in `Auction.sol` (lines 158 to 163 in the `bid` function) will check the root of the tree against the history for membership. If the attacker can forge a merkle tree this will allow non-approved users to submit bids by using the forged tree root.

5. Previous auctioned NFT can be distributed twice

If a previously auctioned NFT is ever auctioned again, the previous winner can distribute the originally auction a second time to regain ownership of the NFT.

The attack works as follows:

- User A wins the NFT in auction 1
- A then calls `distribute(1)` to collect the NFT
- A puts the NFT up for auction in auction 2 and User B wins
- A reclaims the NFT back by calling `distribute(1)` before B can call `distribute(2)` to collect the NFT

Since B won auction 2, they cannot refund their bid (`Auction.sol`, line 195). Furthermore, by A calling `distribute(1)`, the `Auction` contract will no longer own the NFT, so B will no longer be able to claim the NFT as `distribute(2)` will fail during the call to `IERC721.safeTransferFrom` (`Auction.sol`, line 203). A can perform this operation without paying their auction bid again as `distribute` resets the `winningAmt` to 0 (`Auction.sol`, line 205).

This assumes that the NFT itself does not have a malicious contract. If the NFT contract is malicious, this attack could be even more powerful, as A could collude with the NFT contract to allow B's call to `distribute(2)` to succeed without transferring the NFT, so A could reclaim the NFT and have B still pay them the winning bid.

Issues in the protocol

1. Privacy leak in balance note

The cryptographic balance note for a user is defined as $Poseidon(c, a)$. Since the commitment c is public an attacker could perform a dictionary attack over a to figure out the bid amount in an auction. In this kind of attack the attacker would enumerate different values for a and try every single combination until the desired hash is produced.

2. Admin can remove or update commitments

The admin has the capability of modifying the commitments (see `_updateMember` in `AuctionAdmin.sol`). This means that a user whose commitment has been modified could have their funds locked in the auction since they would not be able to withdraw them after the auction.