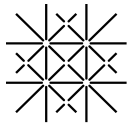

University of Basel

Center for Innovative Finance

Smart Contracts and Decentralized Finance

Sealed Bidding

Prof. Dr. Fabian Schär
University of Basel

Release Ver.: (Local Release)
Version Hash: (None)
Version Date: (None)

Submitting a bid

Input:

- A hashed sealed bid

Tasks:

- Create a new Bid object
- Add the new bid to the sender's list of bids

bid()

```
1 function bid(bytes32 _sealedBid) external payable {
2   Bid memory newBid = Bid({
3     sealedBid: _sealedBid,
4   deposit: msg.value
5   });
6
7   bids[msg.sender].push(newBid);
8 }
```

Data Location

- Notice when creating a new Bid object we need to specify a location for the object.
- All structs and arrays (including strings) can exist in three different locations:
  - `memory`: The object is neither written nor read from the blockchain and only exists in the current scope.
  - `calldata`: Similar to memory, but can only be used for function arguments in external calls and it is non-modifiable.
  - `storage`: Loaded from or written onto the blockchain. These are expensive operations, avoid whenever possible.
- Good heuristic: Use `storage` if you want to load or modify a state variable. Use `memory` otherwise (`calldata` for gas optimization).

The reveal mechanic

Let's think about what the reveal function needs to do:

1. Accept a list of unencrypted bids to reveal.
2. Validate that the length of this list corresponds to the number of committed bids.
3. Check if the reveal period is active.
4. Compute hash values and compare them to commit
5. Ignore fake bids.
6. Update the highest bid and highest bidder if applicable.
7. Handle refunds related to reveal mechanism.

Split the function

For better readability, we split the logic into two parts:

- `updateBid()` handles the bid related checks and refunds.
- `reveal()` handles all the reveal related checks and refunds.

Update Bid

`updateBids()` is similar to the simple auction's bid function:

- The function is internal, meaning it can only be called from the contract itself.
- It returns `true` if the new bid is accepted as the highest bid and `false` otherwise.

Update Bid - Sample code

updateBid()

```
1  function updateBid(address _bidder, uint _bidAmount)
       internal returns (bool success) {
2    if (_bidAmount <= highestBid) {
3      return false;
4    }
5    if (highestBidder != address(0)) {
6      // Refund the previously highest bidder.
7      pendingReturns[highestBidder] += highestBid;
8    }
9    highestBid = _bidAmount;
10   highestBidder = _bidder;
11   return true;
12 }
```

Simple Reveal

Start with a simple version: Only one bid exists:

reveal() - Part I

```
1  function reveal(uint _bidAmount, bool _isLegit,
       string calldata _secret) external {
2    uint refund;
3    Bid storage bidToCheck = bids[msg.sender][0]; //
       Load the first bid of the transaction sender
4    bytes32 hashedInput = generateSealedBid(_bidAmount
       , _isLegit, _secret);
5
6    if (bidToCheck.sealedBid == hashedInput) {
7      // Bid is successfully revealed
8      refund = bidToCheck.deposit;
```

Simple Reveal

reveal() - Part II

```
1       if (_isLegit && bidToCheck.deposit >= _bidAmount
            ) {
2         // Bid is valid
3         bool success = updateBid(msg.sender,
              _bidAmount);
4         if(success) {
5           // Bid is new highest bid
6           refund -= _bidAmount;
7         }
8       }
9       // Prevent re-claiming the same deposit
10      bidToCheck.sealedBid = bytes32(0);
11    }
12    if (refund > 0) {
13      payable(msg.sender).transfer(refund);
14    }
15  }
```

Loops

- We need to iterate over all bids of a user
- Solidity supports most of the control structures known from similar languages such as JavaScript with the usual semantics.
- `for`-loops are typically used to iterate over arrays
- `continue` will jump to the beginning of the next iteration. `break` will end the loop.
- In addition to `for`-loops, `do` and `while` loops are also available.

for()

```
1  // T[] array;
2  for (uint i = 0; i < array.length; i++) {
3    T element = array[i];
4  }
```

Custom Modifiers

- The only thing remaining is to end the auction and to implement time constraints.
- We could do the time constraints similar to the simple auction contract.
- However, if we have the same repeated `require` checks at the start (or end) of functions we can make use of custom modifiers.
- Custom modifiers are a convenient, reusable way to validate inputs to functions.

Custom Modifiers - Code Samples

Only Before

```
1 modifier onlyBefore(uint time) {
2   require(block.timestamp < time, 'too late');
3   _;
4 }
```

Only After

```
1 modifier onlyAfter(uint time) {
2   require(block.timestamp > time, 'too early');
3   _;
4 }
```

Finish the Contract

Add a function to end the auction with the modifier onlyAfter:

Only After

```
1 function auctionEnd() external onlyAfter(revealEnd)
      {
2   require(!hasEnded, 'Auction already ended');
3   emit AuctionEnded(highestBidder, highestBid);
4   hasEnded = true;
5   payable(beneficiary).transfer(highestBid);
6 }
```

Exercise: Update Your SealedBidAuction Contract

Exercise 1

a) Only one bid exists

- Add a function to submit a new bid to your `SealedBidAuction` contract
- Implement a reveal function in your `SealedBidAuction` contract
- Add time constraints to the `bid()`, `reveal()` and `auctionEnd()` functions

b) Extended version: allow for multiple bids

- Extend your contract to handle multiple bids

Hint: You can find all the code components needed for Exercise 1. a) on the previous slides. For Exercise 1. b) you need to extend your contract using a loop.