Vickrey Auction Smart Contract Implementation Report

B285374

1. Design Analysis

1.1 Bid Privacy

By default, the Ethereum blockchain is transparent, as all transaction data is readily available to the public. This is suboptimal for bid privacy, as without any privacy programme, bidders would be able to see each other's bids. This gives an avenue for malicious attacks such as front-running to take place. To ensure bid privacy, I used **commitment schemes**, with a commit and reveal function.

The flow of the commitment scheme is as follows:

1. Off chain, the bidder calls generateCommitment() (see computer programme at end), inputting their bid amount and their nonce. The caller receives a commitment hash in keccak256 form.

2. The bidder calls commitBid() , inputting the auction ID and their commitment hash to commit their bid.

3. The bidder calls revealBid(), inputting their bid amount and nonce from part 1. The contract verifies that the keccak256 hash of their input = the hash commitment from commitBid(). If valid, the bid is counted, otherwise it is rejected.

By following this flow, the commitment cannot reveal information about the bid amount and the bidder cannot change their bid after it is committed. Furthermore, the size of the nonce has to be 32 bytes; brute force attacks become next to impossible due to the large size.

1.2 Seller Payment

For payments, I decided to use a **pull payment pattern**, where users withdraw their own funds instead of getting them sent to them automatically by the contract (push pattern). When the auction is finalised, only the NFT is transferred. To retrieve funds, the bidders must pull their refunds using the withdraw() function, and the seller uses withdrawPayment() to get their payment for the NFT.

By using a pull payment pattern over a push payment pattern:

- A failed withdrawal by one user does not affect everyone (DoS prevention)
- Each user pays gas for their own withdrawal (gas fairness)
- External calls are separated into separate transactions (isolation)
- The auction is finalised before the withdrawals

1.3 Avoiding cheating

1.3.1 Sellers

When a seller calls the createAuction() function, the function ensures the seller owns the NFT, so the seller cannot sell an NFT that is not their own. The NFT transfer can only happen in the finaliseAuction() function, so a seller cannot send an NFT prematurely. Additionally, after bids are completed, there is no way the seller can cancel the auction; the NFT must be transferred.

1.3.2 Bidders

Bidders cannot see other bids (see 2.1), can only bid once, and cannot change their bid after committing (see commitBid() in the programme). This ensures that bidders cannot commit any front running or DoS/griefing attacks. Additionally, bidders cannot lie about their bid amount during their reveal and cannot have a valid bid before depositing at least their bid amount (ensured by revealBid()). This confirms that bidders cannot skew the bid results in their favour without having the appropriate funds.

1.4 Settling ties

If two bids are the same, the first person to call the revealBid() function is deemed the winner. The winner pays the price of the tied bid. This way is the simplest, lacking significant tie-breaking logic. As a result, this is quite gas-efficient. This method is also deterministic and fair, as the transaction order determines the outcome, and all bidders have an equal opportunity to reveal their bid first in the reveal window. It still recognises the second tied bid, however, as that price becomes the value of the auction's second highest bid (the amount the winner will pay).

1.5 Policy on Multiple Bids

Only one bid is allowed per bidder per auction, as implemented in the commitBid() function. This prevents griefing and simplifies logic as bidders cannot spam commitments and multiple bids do not need to be tracked. Gas usage is saved because of the simple logic and lack of griefing. Additionally, this levels the playing field as each bidder has equal odds; bidders simply must commit their best offer without considering other factors.

1.6 Data Structures

struct auction { address seller; address nftContract; uint tokenId; uint reservePrice; uint biddingEnd; uint revealEnd; address highestBidder; uint highestBid; uint secondHighestBid; uint finalPrice; bool finalised; bool paymentWithdrawn; }

This structure uses timestamps for biddingEnd and revealEnd to make it easier for the users. The highest bid and second highest bid are tracked separately for easier calculations and status flags such as finalised and paymentWithdrawn prevent double execution.

struct bid { bytes32 commitment; uint deposit; bool revealed; uint bidAmount; bool withdrawn;}

This structure only has five fields(optimised for gas), with state flags *revealed* and *withdrawn* for preventing double execution. Additionally, the deposit is separate from the bid, for security reasons (attackers cannot guess the bid from the deposit).

mapping(uint => auction) public auctions; mapping(uint => mapping(address => bid)) public bids; mapping(uint => address[]) private bidders;

These mappings allow for gas efficiency (quicker lookup time for auctions), scalability (only stores actual bids) and easier accessing through indexing by auctionID and address.

2. Gas Analysis

2.1 Cost

	Seller	Bidder 1 (460a)	Bidder 2 (D95d)
createAuction()	1.5Gwei	-	-
commitBid()	-	1.5Gwei	1.5Gwei
revealBid()	-	1.5Gwei	1.5Gwei
finaliseAuction()	1.5Gwei	-	-
withdrawSeller()	1.5Gwei	-	-
withdraw()	-	1.5Gwei	1.5Gwei
Overall gas:	4.5 Gwei	4.5Gwei	4.5Gwei

TOTAL GAS COST(INTERACTION WITH 2 BIDDERS*): 13.5 Gwei

**note: gas usage for interactions will increase by approximately 4.5Gwei per bidder.*

TOTAL GAS COST(DEPLOYMENT): 28.215Gwei

2.2 Gas Equality

Overall, the bid process is fair in terms of gas consumption. Each bidder must call the same commitBid(), revealBid(), and withdraw() functions, meaning approximately the same amount of gas is used for all bidders. The only extra gas cost comes from finaliseAuction() (only called once, whoever calls it pays for it). This is still somewhat fair, however, as whoever receives the most benefit from the bid tends to call finaliseAuction(), as they want to get their benefits as soon as possible.

2.3 Techniques

Initially, I used a for loop to refund users but undid that to prevent high gas usage situations because of potential DoS attacks. By changing the refund system to a pull technique, I prioritised fairness over full gas efficiency, so only one user (likely the seller) would not be paying for everyone's refunds. There are also not many storage writes as I opted to use states and update them, when necessary, instead of storing values. Additionally, I calculated refunds while the program is running instead of storing it, and bidders could only bid once (using less storage space). By saving storage space, the consumption of gas reduced significantly.

2.4 Sepolia Hashes

Address of Contract: 0xe036F05627f6ffaE2845A33a2f949e2EBD8795D4

createAuction Transaction hash:
0x83a675043d80c718b2e54e8ec4faf606d6f66ed20a322a8e918978106c709f15

commitBid Transaction hash (addr ending in 460a):
0x626de3867bb9019f8abd4d9ab2e3f583bfd37a4f09e044808f307934abc1eed2

commitBid Transaction hash (addr ending in D95d):
0x85bfb0a01dcce96b549abc2a2acf90f4b7d5ebcc8f84d46d6cc86f0557441eed

revealBid Transaction hash (addr ending in D95d):
0x9ce892fafbefa924058e3a7c790a20346386007c54047460c43da103b8dd48b2

revealBid Transaction hash (addr ending in 460a):
0x829ac924b183c429cc14ee2061043ec9d6c46acb788467c77036849e60c5ae99

finaliseAuction Transaction hash:
0x1785969bbd7b2b1ac1b9387821b0ad8c44669742129f1f9b2433eb708a506595

withdrawSeller Transaction hash:
0xc76bbb1deda922eb3dd19850e4c88e69f5416186591e6edf987b42ba38e0c843

withdraw Transaction hash (addr ending in 460a):
0xd0b30f2f9304a80a009fb7bd32228ab0a882adbf46ea3f1625f1d1711111927d

withdraw Transaction hash (addr ending in D95d):
0x770527ebf2537bdc12fa8522f71aba3cf67099f39a31e543848b2b62760948f6

3. Security Analysis

3.1 Vulnerabilities

3.1.1 Re-entrancy Attack

Vulnerability: withdraw()/withdrawSeller() is called repeatedly to drain funds

Mitigation: To avoid this problem, the Checks-Effects-Interactions pattern is used. In withdraw(), the conditions are checked first (auction finalised, bid deposit exists, bid is not withdrawn). Then, the bid.withdrawn state is set to true, and then the transfer takes place. This ensures that once a bidder withdraws their funds, a re-entry call would fail. The same logic is the case for withdrawSeller().

3.1.2 Front Running Attack

Vulnerability: Another bidder extracts the bid amount of a committed bid and places a higher bid to win the auction.

Mitigation: Commitment schemes are used to hide bid amounts. The commitBid() function only has the commitment hash, and it is near impossible to get the bid value from the hash, especially due to the size of the nonce in the hash. When bids finally get revealed, the malicious user is unable to bid, as bids can only be revealed once the bidding period is over.

3.1.3 Denial of Service Attack

Vulnerability: A malicious attacker may commit many bids and overload a loop of bidders to process refunds, leading to the finaliseAuction() function being blocked.

Mitigation: Initially, there was a loop in the finaliseAuction() function to refund bidders and the seller. Realizing the issue, I removed the loop and created a pull payment pattern, where the bidders calls withdraw() and the seller calls withdrawSeller(). This means all bids are managed in isolation and there is no infinite loop, preventing any overflow.

3.1.4 Griefing Attack

Vulnerability: An attacker can repetitively submit the same commitment hashes of legitimate users, being a hindrance to the contract and users.

Mitigation: Only one commitment per address is allowed as shown in commitBid(), so an attacker would not be able to do this. Additionally, if the attacker manages to commit the same hash, they cannot reveal it as they do not know the bid value or nonce.

3.1.5 Overflow Attack

Vulnerability: Arithmetic overflows may happen, leading to incorrect calculations.

Mitigation: Solidity 0.8+ is used with inbuilt overflow protection, reverting arithmetic issues

3.1.6 Seller Backing Out

Vulnerability: The seller backs out after seeing the bids to avoid the sale, which is unfair to the bidders who spent gas on the auction.

Mitigation: There is no cancellation function for the seller, once the seller creates the auction, the seller is committed to transferring the NFT.

3.1.7 Malicious NFT Contract

Vulnerability: The seller gives the address of a fake NFT contract that would fail during the transfer.

Mitigation: The transfer can only happen after the payment is secured, and the auction state is marked as finalised before transfer. This ensures that the winner and seller can still withdraw funds. However, the transfer failure will result in the winner not receiving an NFT, the winner would have to contact the seller about this.

3.2 Remaining Risks and Limitations

3.2.1 Bidder Forgets Nonce

Risk: The bidder forgets their nonce, so they cannot reveal their bid. The bidder would then lose the auction.

Result: Not a security risk, and the bidder can receive their funds back by calling the withraw() function. Mildly inconvenient for the bidder.

3.2.2 NFT Approval Missing

Risk: The seller fails to approve the auction before or during the auction, so the NFT transfer fails. This leads to finaliseAuction() reverting and bidder deposits being locked.

Result: Not a serious security risk as it is not exploitable by a malicious user, provided the seller cooperates. I did not do the approval check at creation to allow flexibility and a better user experience for the seller.

3.2.3 Unclaimed Funds

Risk: The users of the contract do not call withdraw()/withdrawSeller() and the funds remain in the contract.

Result: Not a significant security risk as the funds remain claimable and no one else but the specific user can claim them. It may be inconvenient for a user but ensures security against DoS/griefing.

4. Design Trade-offs

4.1 Security & Gas Efficiency

By using a pull payment pattern, gas cost increases per person. However, the security of the program is drastically approved. Increasing the average gas usage to accommodate pull patterns significantly improves security and gas fairness. A push payment pattern would contain a loop, which opens it up to DoS attacks and allows for easy re-entrancy. Additionally, the last bidder would pay the most versus the pull pattern, where everyone pays equally to withdraw their funds.

4.2 Privacy & Transparency

In the programme, the commitment scheme hides bids, but only temporarily. If the bids were always hidden, the code would be extremely complex as it would be difficult to compare hidden bids. In this trade-off, conditions are put in place to only hide the bids at critical times, avoiding front-running while conserving simplicity and gas.

4.3 Flexibility & Simplicity

This programme only allows one bid per user, which simplifies the programme logic but reduces flexibility. The simplicity (lack of storing and computing) significantly reduces gas costs. Furthermore, this reduces the threat of DoS/griefing attacks as one user cannot “spam” bids.

4.4 User Experience & Decentralization

By allowing anyone to finalise security risks are mitigated but it may lead to delays (users may not immediately call finaliseAuction()). If only the seller could finalise, they could individually choose not to finalise the auction (seller griefing). This would not be the case if anyone could call finaliseAuction(). Additionally, users should generally be inclined to finalise the auction quickly, as they would want their funds back ASAP.

5. Peer Contract Analysis

5.1 Strengths in Security:

The programme overall is quite secure. There is a modifier specifically set to prevent re-entry attacks throughout the ETH transfer, along with a great use of the checks-effects-interactions

pattern. This ensures strong protection against malicious behaviour, especially re-entry attacks. Commitment schemes are used as well (just like my code) and the receive() function reverts direct transfers, rejecting accidental ETH. A secure feature the program has that is unique is that it ensures that the NFT transfer is approved before creating the auction.

```
// This will protect contract from re-entrancy attacks
uint256 private _entered =1;
modifier nonReentrant() {
    require(_entered == 1, "reentrancy");
    _entered = 2; _;
    _entered =1;
}
```

```
//This is the withdraw function for the winers to get the extra change over the clearing price
function withdraw() external nonReentrant {
   uint256 amt = refunds[msg.sender];
   require(amt > 0, "nothing");
   refunds[msg.sender] = 0;
   (bool ok, ) = payable(msg.sender).call{value: amt}("");
   require(ok, "withdraw failed");
   emit Refunded(msg.sender, amt);
}
```

Left top: Re-entry modifier

Left bottom: Checks-Effects-Interactions pattern used in withdraw()

5.2 Weaknesses

In the programme, the payment happens before the NFT transfer. This means that if the transfer is unable to happen, the auction would become stuck and all stakeholders would lose their funds. A malicious seller could run an attack such that they get the payment first and ensure the transfer fails, so they can keep the payment without transferring the NFT. Additionally, the finalisation of the contract needs two functions (finalise() and settle()), which means higher gas consumption than required.

```
function settle(uint256 id) external exists(id) nonReentrant {
    Auction storage A = auctions[id];
    require(A.finalised, "not finalised");
    require(!A.settled, "settled");

    IERC721 token = IERC721(A.nft);
    if (A.winner == address(0)) {
        token.safeTransferFrom(address(this), A.seller, A.tokenId);
    } else {
        token.safeTransferFrom(address(this), A.winner, A.tokenId);
    }

    if (A.potForSeller > 0) {
        (bool ok, ) = payable(A.seller).call{value: A.potForSeller}("");
        require(ok, "pay seller failed");
    }

    A.settled = true;
    emit BidSettled(id, A.seller, A.winner, A.potForSeller);
}
```

Above: The settle() function, which allows for the transfer to happen before the payment.

5.3 Gas

Contract Deployment	1,378,513
startAuction	221,927
commitBid	50,547
revealBid (first)	146,724
revealBid (subsequent)	157,322
finalize	157,566
settle	90,953
withdrawLoser	41,119
withdraw	32,292

Overall, the bidders are treated quite equally in terms of the gas usage, and the programme is mostly well-designed in terms of gas. Having both finalize() and settle() instead of just one master finalisation function along with having the escrow in-built increases gas usage.

5.4 Peer contract Details

ClassNFT Contract: 0x1546Bd67237122754D3F0cB761c139f81388b210
Token ID: 435
Mint Transaction:
0xdcf02443e30bf4eab1b735bc194894a0949e4344c739fba6e725ceb5ddf40e2c
Auction Contract: 0x0c932e5b5b0641643a8354664b4847bbe642ba6d
Deployment Tx:
0xde3746d567cc9b2456d21a84690b3cd97140c96720607bd3631d51feace3a784
NFT Approval:
0x8601d9091e5309a93877c2ef1daf4dfd833fa7ba82e1a4f0a01a90bb5873f2bf
Start Auction:
0x577ea42f03b9fc34eb7041105c144f2c6f4657f71f53a3f4335a32a4068881b6
Bidder 1 Commit:
0x1bce9ffc41be538d9617850c8e82f781bebff41c549082bb1de43d7b0fd6b3b2
Bidder 2 Commit:
0xb4b36efb1696eaabfc0ef6d88e73c061f1eb5caff4ae4d953cf7494d26a86d91
Bidder 1 Reveal:
0xae45f84c47dd6e6c15d856c037298b4f722c817356f196f924ef49d73b7b3712
Bidder 2 Reveal:
0x7dfa32f0acdaaf1a2f98197abb890f49bac79c7a5b14777e6889e4e84a05c64c
Finalize:
0xd7281d1cded63e9be756394afd2e6daa6018311094dd11c0ef108418a17e643e
Settle:
0xe32166f53b5214b440cf5df13438007d1db071b392bdb88d60b5c4048826f2b6
Withdrawals:

Bidder 1:
0xed04062e77b6e98c9fddef23b3382fcc3e67b355b2c2533916db64296b9233f4
Bidder 2:
0x057a2159b6d4c22458baf6573e59b1da6f6909a8372a26e43dc0127442119185

6. My program:

```
/* PREAMBLE
    This programme is a Smart Contract for the auctioning of NFT tokens, using the Vickrey Auction method(sealed-bid, second price),
    The programme is made to ensure maximum security, mainly using commitment schemes to keep data private and prevent attacks(more in the report).

    Flow of the programme:
    1. The owner of the NFT calls an auction using the createAuction function, declaring their NFT contract and token's ID.
       In the call, they declare the NFT contract and token ID, while specifying the auction's reserve price, bidding duration, and reveal duration.
       The function creates an ID for the specific auction and returns it, so the seller is aware of the auction ID
    2. The bidder uses the getCurrentAuctionId function to get access to the most current(and likely ongoing) auction
    3. The bidder uses the getAuction function with the auction ID to access information about the auction they want to bid in, including the reserve price.
    4. The user decides their bid value and creates their hash commitment off the chain, using the generateCommitment function.
    5. The bidder uses the commitBid function to send their committed bid for the specific auction, while sending their deposit that should be higher than the bid.
    6. The bidder uses the revealBid function to reveal their bid, inputting their bid value and nonce (ONLY AFTER BIDDING ENDS).
       The programme will use inputted data to verify bidder information, and set the highest and second-highest bidders.
    7. Anyone can call the finaliseAuction function ONCE REVEALING IS COMPLETE.
       The NFT is transferred from the seller to the highest bidder at the price of the second highest bid.
    8. The seller receives their funds for the NFT by calling the withdrawSeller function.
```

```
    9. The winner receives their refund (deposit-price of NFT) and the losers obtain their bids by calling the withdraw function.



    Code is indexed into numeric indexes to ensure comprehension.
*/

//0: ENSURE CORRECT SOLIDITY VERSION IS USED, INTERFACE INCLUDED FOR ERC-721

//SPDX-License-Identifier-MIT
pragma solidity ^0.8.0;

//Interface ERC721
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";

contract VickreyAuction {

    //1: STRUCTS, NECESSARY VARIABLES

    //1.1 Struct for auctions
    struct auction {
        address seller;
        address nftContract;
        uint tokenId;
        uint reservePrice;
        uint biddingEnd;
        uint revealEnd;
        address highestBidder;
        uint highestBid;
        uint secondHighestBid;
        uint finalPrice;
        bool finalised;
        bool paymentWithdrawn;
    }

    //1.2 Struct for bids
    struct bid {
        bytes32 commitment;
        uint deposit;
        bool revealed;
        uint bidAmount;
        bool withdrawn;
    }
```

```
65.
66.     //1.3 Auction counter (Used for Auction ID)
67.     uint public auctionCounter;
68.
69.     //2. STORAGE MAPPING
70.     mapping(uint => auction) public auctions;
71.     mapping(uint => mapping(address => bid)) public bids;
72.     mapping(uint256 => address[]) private bidders;
73.
74.
75.     //3. EVENTS
76.
77.     //3.1 Logs relevant data when an auction is created
78.     event auctionCreated (
79.         uint indexed auctionId,
80.         address indexed seller,
81.         address nftContract,
82.         uint tokenId,
83.         uint reservePrice,
84.         uint biddingEnd,
85.         uint revealEnd
86.     );
87.
88.     //3.2 Logs relevant data when a bid is committed
89.     event bidCommitted (
90.         uint indexed auctionId,
91.         address indexed bidder,
92.         bytes32 commitment,
93.         uint deposit
94.     );
95.
96.     //3.3 Logs relevant data when a bid is revealed
97.     event bidRevealed (
98.         uint indexed auctionId,
99.         address indexed bidder,
100.            uint bid
101.        );
102.
103.        //3.4 Logs relevant data when the auction is finalised
104.        event auctionFinalised (
105.            uint indexed auctionId,
106.            address winner,
107.            uint winningBid,
108.            uint pricePaid
109.        );
```

```
110.
111.         //3.5 Logs relevant data when the withdrawal is ready
112.         event WithdrawalReady (
113.             address user,
114.             uint amount
115.         );
116.
117.         //3.6 Logs relevant data when the withdrawals are performed
118.         event WithdrawalPerformed (
119.             address indexed user,
120.             uint amount
121.         );
122.
123.         //3.7 Logs relevant data when the seller is paid
124.         event SellerPaid (
125.             address indexed seller,
126.             uint amount
127.         );
128.
129.         //4. MODIFIERS
130.
131.         //4.1 Ensures auction is valid
132.         modifier auctionExists (uint auctionId) {
133.             require(auctionId < auctionCounter, "The auction does not
    exist.");
134.             _;
135.         }
136.
137.         //4.2 Ensures that the seller is the one sending the message
138.         modifier onlySeller (uint auctionId) {
139.             require(msg.sender == auctions[auctionId].seller, "This is
    not the seller.");
140.             _;
141.         }
142.
143.         //5. CORE FUNCTIONS
144.
145.         //5.1 Create a new auction, return the auction identifier
146.         function createAuction(address nftContract, uint tokenId, uint
    reservePrice, uint biddingDuration, uint revealDuration)
147.         external returns (uint) {
148.
149.             //5.1.1 ensure valid inputs
150.             require(nftContract != address(0), "This NFT contract is
    invalid.");
```

```
151.             require(biddingDuration > 0, "The bidding duration must be >
   0.");
152.             require(revealDuration > 0, "The reveal duration must be >
   0.");
153.
154.             //5.1.2 ensure seller owns the NFT
155.             IERC721 nft = IERC721(nftContract);
156.             require(nft.ownerOf(tokenId) == msg.sender, "This is not the
   NFT owner.");
157.
158.             //5.1.3 initiate auction variables
159.             uint auctionId = auctionCounter++;
160.             uint biddingEnd = block.timestamp + biddingDuration;
161.             uint revealEnd = biddingEnd + revealDuration;
162.
163.             //5.1.4 insert relevant inputs into Auction struct in
   auctions array, setting 0/false to unknowns
164.             auctions[auctionId] = auction({
165.                 seller: msg.sender,
166.                 nftContract: nftContract,
167.                 tokenId: tokenId,
168.                 reservePrice: reservePrice,
169.                 biddingEnd: biddingEnd,
170.                 revealEnd: revealEnd,
171.                 highestBidder: address(0),
172.                 highestBid: 0,
173.                 secondHighestBid: 0,
174.                 finalPrice: 0,
175.                 finalised: false,
176.                 paymentWithdrawn: false
177.             });
178.
179.             //5.1.5 log Auction Created event into the transaction
180.             emit auctionCreated(
181.                 auctionId,
182.                 msg.sender,
183.                 nftContract,
184.                 tokenId,
185.                 reservePrice,
186.                 biddingEnd,
187.                 revealEnd
188.             );
189.
190.             return auctionId;
191.         }
```

```
192.
193.         //5.2 Commit to the bid
194.         function commitBid(uint auctionId, bytes32 commitment)
195.         external payable auctionExists(auctionId) {
196.
197.             //5.2.1 get access to the specific auction in storage
198.             auction storage auction = auctions[auctionId];
199.
200.             //5.2.2 ensure bid requirements are fulfilled
201.             require(block.timestamp < auction.biddingEnd, "Bidding
    ended, too late.");
202.             require(msg.value > 0, "Deposit not sent. Send a deposit >=
    than your bid.");
203.             require(commitment != bytes32(0), "Invalid committment");
204.
205.             //5.2.3 get access to the bidder's bid for the auction
206.             bid storage bid = bids[auctionId][msg.sender];
207.
208.             //5.2.4 ensure no one recommits (preventing front-running)
209.             require(bid.commitment == bytes32(0), "Bid already
    submitted. Only one bid allowed.");
210.
211.             //5.2.5 initializing values to relevant variables
212.             bid.commitment = commitment;
213.             bid.deposit = msg.value;
214.             bid.revealed = false;
215.             bid.bidAmount = 0;
216.             bid.withdrawn = false;
217.             bidders[auctionId].push(msg.sender);
218.
219.             //5.2.6 logging event into transaction log
220.             emit bidCommitted(
221.                 auctionId,
222.                 msg.sender,
223.                 commitment,
224.                 msg.value
225.             );
226.         }
227.
228.         //5.3 Reveal your bid
229.         function revealBid(uint auctionId, uint bidAmount, bytes32
    nonce)
230.         external auctionExists(auctionId) {
231.
232.             //5.3.1 access auction and bid from storage
```

```
233.             auction storage auction = auctions[auctionId];
234.             bid storage bid = bids[auctionId][msg.sender];
235.
236.             //5.3.2 ensure variables/values needed are valid
237.             require(block.timestamp >= auction.biddingEnd, "Bidding in
    progress.");
238.             require(block.timestamp < auction.revealEnd, "Reveal period
    complete.");
239.             require(bid.commitment != bytes32(0), "No commitment
    found.");
240.             require(!bid.revealed, "Bids already revealed.");
241.             require(bidAmount <= bid.deposit, "Bid exceeds deposit
    given.");
242.
243.             //5.3.3 verify the commitment
244.             bytes32 computedCommitment =
    keccak256(abi.encodePacked(bidAmount, nonce));
245.             require(computedCommitment == bid.commitment, "Invalid.");
246.
247.             //5.3.4 complete bidding process
248.             bid.revealed = true;
249.             bid.bidAmount = bidAmount;
250.
251.             //5.3.5 update the highest and second highest bids
252.             if(bidAmount >= auction.reservePrice) {
253.                 if(bidAmount > auction.highestBid) { //new highest bid
254.                     auction.secondHighestBid = auction.highestBid;
255.                     auction.highestBid = bidAmount;
256.                     auction.highestBidder = msg.sender;
257.                 } else if (bidAmount > auction.secondHighestBid) { //new
    second highest bid
258.                     auction.secondHighestBid = bidAmount;
259.                 }
260.             }
261.
262.             //5.3.6 add event to transaction log
263.             emit bidRevealed(
264.                 auctionId,
265.                 msg.sender,
266.                 bidAmount);
267.         }
268.
269.         //5.4. Finalise auctions after bids are revealed
270.         function finaliseAuction(uint auctionId) external
    auctionExists(auctionId){
```

```
271.
272.             //5.4.1 access auction from storage
273.             auction storage auction = auctions[auctionId];
274.
275.             //5.4.2 ensure all parameters are met for this function
276.             require(block.timestamp >= auction.revealEnd, "Reveal phase
   incomplete.");
277.             require(!auction.finalised, "Auction already finalised.");
278.
279.             //5.4.3 officially finalise the auction
280.             auction.finalised = true;
281.
282.             //5.4.4 determine the winner and the price to pay
283.             if(auction.highestBid >= auction.reservePrice) {
284.                 uint price;
285.                 if (auction.secondHighestBid >= auction.reservePrice){
286.                     price = auction.secondHighestBid;
287.                 } else {
288.                     price = auction.reservePrice;
289.                 }
290.                 auction.finalPrice = price;
291.
292.                 //5.4.4.1 winner determined
293.                 address winner = auction.highestBidder;
294.
295.                 //5.4.4.2 transfer the NFT
296.                 IERC721(auction.nftContract).safeTransferFrom(
297.                     auction.seller,
298.                     winner,
299.                     auction.tokenId);
300.
301.                 //5.4.4.3 add event to transaction log
302.                 emit auctionFinalised(
303.                     auctionId,
304.                     winner,
305.                     auction.highestBid,
306.                     price);
307.             } else {
308.                 //5.4.4.4 no valid bids in auction
309.                 emit auctionFinalised(auctionId, address(0), 0, 0);
310.             }
311.         }
312.
313.         //5.5 Withdraw (pull) funds for the bidder
```

```
314.         function withdraw(uint auctionId) external
   auctionExists(auctionId) {
315.
316.             //5.5.1 ensure auction is finalised
317.             auction storage auction = auctions[auctionId];
318.             require(auction.finalised, "Auction not finalized");
319.
320.             //5.5.2 ensure no malicious behaviour with bids
321.             bid storage bid = bids[auctionId][msg.sender];
322.             require(bid.deposit > 0, "No deposit");
323.             require(!bid.withdrawn, "Already withdrawn");
324.
325.             uint refundAmount;
326.
327.             //5.5.3 set amount to be refunded for all bidders
328.             if (msg.sender == auction.highestBidder &&
   auction.highestBid >= auction.reservePrice) {
329.                 if(bid.deposit > auction.finalPrice) {
330.                     refundAmount = bid.deposit - auction.finalPrice;
331.                 } else {
332.                     refundAmount = 0;
333.                 }
334.             } else {
335.                 refundAmount = bid.deposit;
336.             }
337.
338.             //5.5.4 mark the bid as withdrawn
339.             bid.withdrawn = true;
340.
341.             //5.5.5 transfer the refund to the bidder
342.             (bool success, ) = msg.sender.call{value: refundAmount}("");
343.             require(success, "Transfer failed");
344.
345.             //5.5.6 add event to transaction log
346.             emit WithdrawalPerformed(msg.sender, refundAmount);
347.         }
348.
349.         // 5.6 Withdrawal function for the seller
350.         function withdrawSeller(uint256 auctionId) external
   auctionExists(auctionId) onlySeller(auctionId){
351.             auction storage auction = auctions[auctionId];
352.             require(auction.finalised, "Auction not finalized");
353.             require(!auction.paymentWithdrawn, "Already withdrawn");
354.             require(auction.highestBid >= auction.reservePrice, "No
   sale");
```

```
355.
356.             auction.paymentWithdrawn = true;
357.             uint payment = auction.finalPrice;
358.
359.             (bool success, ) = msg.sender.call{value: payment}("");
360.             require(success, "Transfer failed");
361.
362.             emit SellerPaid(msg.sender, payment);
363.         }
364.
365.         //6. GET FUNCTIONS
366.
367.         //6.1 get auction details
368.         function getAuction(uint auctionId) external view
    auctionExists(auctionId)
369.         returns(
370.             address seller,
371.             address nftContract,
372.             uint tokenId,
373.             uint reservePrice,
374.             uint biddingEnd,
375.             uint revealEnd,
376.             address highestBidder,
377.             uint highestBid,
378.             uint secondHighestBid,
379.             bool finalised
380.         ) {
381.             auction storage auction = auctions[auctionId];
382.             return(
383.                 auction.seller,
384.                 auction.nftContract,
385.                 auction.tokenId,
386.                 auction.reservePrice,
387.                 auction.biddingEnd,
388.                 auction.revealEnd,
389.                 auction.highestBidder,
390.                 auction.highestBid,
391.                 auction.secondHighestBid,
392.                 auction.finalised
393.             );
394.         }
395.
396.         //6.2 get current auction ID
397.         function getCurrentAuctionID() external view returns(uint
    auctionID){
```

```
398.             return (auctionCounter - 1);
399.         }
400.
401.         //7. HASH FUNCTION REQUIRED FOR THE COMMITMENT (BIDDER MUST DO
   THIS OFF-CHAIN, and use the hash to commit their bid)
402.         function generateCommitment(uint bidAmount, bytes32 nonce)
403.         external pure returns(bytes32){
404.                 return keccak256(abi.encodePacked(bidAmount, nonce));
405.         }
406.     }
407.
```