Energy Bidding Market

NOVA Information Management School

Smart Contract Security Review

Report Prepared By:

José Garção

Márcio Romão

April 4, 2025

Contents

1. Disclaimer

The team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

2. Severity Classification

Impact / Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

2.1 Impact

- **High** - Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- **Medium** - Global losses <10% or losses to only a subset of users, but still unacceptable.
- **Low** - Losses will be annoying but bearable - applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

2.2 Likelihood

- **High** - Almost certain to happen, easy to perform, or not easy but highly incentivized.
- **Medium** - Only conditionally possible or incentivized, but still relatively likely.
- **Low** - Requires stars to align, or little-to-no incentive.

2.3 Action Required

- **Critical** - Must fix
- **High** - Must fix
- **Medium** - Should fix
- **Low** - Could fix
- **Informational** - Optional fix

3. Executive Summary

Over the course of 14 days, NOVA Information Management School engaged with the team to review the Energy Bidding Market protocol.

3.1 People Involved

- José Garção | Github Profile
- Márcio Romão | Github Profile

3.2 Summary

Version	1.0
Project Name	Energy Bidding Market
Repository	UniformBiddingMarket
Commit Hash	0d4b900
Language	Solidity
Platform	Nova Cidade Chain
Timeline	March 24 to April 4

3.3 Scope of Work

- `src/EnergyBiddingMarket.sol`

3.4 Findings Summary

Severity	Count	Fixed	Acknowledged
Critical	3	0	0
High	2	0	0
Medium	0	0	0
Low	5	0	0
Informational	8	0	0
Total	**18**	**0**	**0**

4. Findings

In this section, we present the details of each finding. To run each proof of concept, please add the following `PoC.t.sol` file to the `test/` folder:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {EnergyBiddingMarket} from "../src/EnergyBiddingMarket.sol";
import {Upgrades} from "openzeppelin-foundry-upgrades/Upgrades.sol";

contract PoC is Test {
    // Contracts
    EnergyBiddingMarket market;
    // Actors
    address OWNER = makeAddr("owner");
    address BIDDER = makeAddr("bidder");
    address ASKER = makeAddr("asker");

    function setUp() public {
        skip(180 days);
        market = EnergyBiddingMarket(
            Upgrades.deployUUPSProxy(
                "EnergyBiddingMarket.sol:EnergyBiddingMarket", abi.encodeWithSignature("initialize(address)", OWNER)
            )
        );
        deal(BIDDER, 100 ether);
    }
}
```

For each test, add the relevant function to the file above and run this command (replace `[function-name]` with the corresponding function name):

```
forge clean && forge build && forge test --mt [function-name] -vv
```

> **Note:** Some findings may have additional instructions for running the corresponding proofs of concept. Please follow these instructions where provided.

4.1 Critical Findings

4.1.1 Incorrect sorting in `sortBids` leads to a flawed clearing price and erroneous settlement between bids and asks

Description

The `sortBids` function is intended to sort bids by price in descending order. However, the current implementation is flawed, causing bids to be incorrectly sorted. As a result, both the settlement between bids and asks and the clearing price are inaccurate.

Context

- EnergyBiddingMarket.sol#L387-L421

Impact

High. This issue disrupts the core functionality of the protocol, leading to incorrect settlement between bids and asks and an inaccurate clearing price for energy trades.

Likelihood

High. Due to the flawed sorting algorithm, this issue will always occur.

Proof of Concept

```
function test_IncorrectSorting() public {
    // Prepare hour to place bids
    uint256 hour = marketHarness.getCurrentHourTimestamp() + 3600;

    // Prepare energy and eth amounts for each bid
    uint256 numberOfBids = 5;
    uint256[] memory energyAmounts = new uint256[](numberOfBids);
    uint256[] memory ethAmounts = new uint256[](numberOfBids);

    // Populate energy amounts for each bid
    energyAmounts[0] = 5791;
    energyAmounts[1] = 8472;
    energyAmounts[2] = 953;
    energyAmounts[3] = 8403;
    energyAmounts[4] = 9565;

    // Populate eth amounts for each bid
    ethAmounts[0] = 479008935626859662;
    ethAmounts[1] = 276139232672438773;
    ethAmounts[2] = 743742146016760527;
    ethAmounts[3] = 33642988462095454;
    ethAmounts[4] = 350037435968563937;

    // Places bids
    vm.startPrank(BIDDER);
    for (uint256 i; i < numberOfBids; ++i) {
        marketHarness.placeBid{value: ethAmounts[i]}(hour,
↪   energyAmounts[i]);
    }
    vm.stopPrank();

    // Log bids before sorting
    EnergyBiddingMarket.Bid[] memory unsortedBids =
↪   marketHarness.getBidsByHour(hour);
    console.log("# Unsorted Bids");
    _logBidPrices(unsortedBids);

    // Sort bids
    marketHarness.sortBidsExposed(hour);

    // Log bids after sorting
    EnergyBiddingMarket.Bid[] memory sortedBids =
↪   marketHarness.getBidsByHour(hour);
    console.log("# Sorted Bids");
    _logBidPrices(sortedBids);

    // Check if bids are indeed sorted
    bool isSorted = true;
    for (uint256 k = 1; k < numberOfBids; ++k) {
        if (sortedBids[k].price > sortedBids[k - 1].price) {
            isSorted = false;
            break;
        }
    }

    // Assert that bids are not sorted correctly
    assertFalse(isSorted);
}
```

Logs

```
[PASS] test_IncorrectSorting() (gas: 535226)
Logs:
  # Unsorted Bids
  Price of bid #0 : 0.000082716100090979 ETH
  Price of bid #1 : 0.00003259433813414 ETH
  Price of bid #2 : 0.000780421979031228 ETH
  Price of bid #3 : 0.000004003687785564 ETH
  Price of bid #4 : 0.000036595654570681 ETH

  # Sorted Bids
  Price of bid #0 : 0.00003259433813414 ETH
  Price of bid #1 : 0.000004003687785564 ETH
  Price of bid #2 : 0.000082716100090979 ETH
  Price of bid #3 : 0.000036595654570681 ETH
  Price of bid #4 : 0.000780421979031228 ETH

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.08s
 ↪  (572.30µs CPU time)
```

Instructions

1. Add the following `EnergyBiddingMarketHarness.sol` file to the `test/` folder:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {EnergyBiddingMarket} from "../src/EnergyBiddingMarket.sol";

contract EnergyBiddingMarketHarness is EnergyBiddingMarket {
    // Function to expose the internal sortBids
    function sortBidsExposed(uint256 hour) external {
        sortBids(hour);
    }
}
```

2. Add the following code to the `PoC.t.sol` file:

```
  // SPDX-License-Identifier: UNLICENSED
  pragma solidity ^0.8.13;


  ...
```

```
+ import {EnergyBiddingMarketHarness} from
↪  "./EnergyBiddingMarketHarness.sol";

  contract PoC is Test {
      // Contracts
      EnergyBiddingMarket market;
      EnergyBiddingMarketHarness marketHarness;

      ...

      function setUp() public {
          ...
+         marketHarness = EnergyBiddingMarketHarness(
+             Upgrades.deployUUPSProxy(
+
↪  "EnergyBiddingMarketHarness.sol:EnergyBiddingMarketHarness",
+                 abi.encodeWithSignature("initialize(address)",
↪  OWNER)
+             )
+         );
          ...
      }

      ...

+     function _logBidPrices(EnergyBiddingMarket.Bid[] memory bids)
↪  private pure {
+         for (uint256 i; i < bids.length; ++i) {
+             console.log("Price of bid #%d : %18e ETH", i,
↪  bids[i].price);
+         }
+         console.log("");
+     }
  }
```

Recommendation

Ensure that the `sortBids` function correctly sorts bids in descending order by price to prevent settlement errors and incorrect clearing prices.

4.1.2 The `_clearMarket` function does not check for canceled bids causing them to be matched anyway

Description

The `_clearMarket` function is intended to settle bids and asks for a specific hour. However, it does not account for bids that have been canceled. As a result, canceled bids will still be matched, causing askers to sell energy without being able to withdraw the corresponding `ETH` amount from the contract.

Context

- EnergyBiddingMarket.sol#L304-L383

Impact

High. This issue causes askers to settle with canceled bids and, consequently, prevents them from receiving the corresponding `ETH` amount back.

Likelihood

High. Due to the absence of this check, canceled bids will always be matched.

Proof of Concept

```
function test_CanceledBidMatched() public {
    // Compute hour to bid
    uint256 hour = market.getCurrentHourTimestamp() + 3600;

    // Bidder places a bid, cancel it, and claims back the ETH
    vm.startPrank(BIDDER);
    market.placeBid{value: 1 ether}(hour, 100);
    market.cancelBid(hour, 0);
    market.claimBalance();
    vm.stopPrank();

    // Warp to the timestamp where askers begin to ask
    vm.warp(hour);

    // Asker places an ask to fully match the bid
    vm.prank(ASKER);
    market.placeAsk(100, ASKER);

    // Skip 1 hour so that we can clear the market and settle the
     ↪  orders
    skip(3600);

    // Clear the market
    market.clearMarket(hour);

    // Assert that the asker has balance, meaning canceled bid was matched with this ask
    assertEq(market.balanceOf(ASKER), 1 ether);

    // Assert that transaction reverts when asker tries to claim the balance
    vm.expectRevert();
    vm.prank(ASKER);
    market.claimBalance();

    // Log eth balance of the contract
    console.log("ETH BALANCE OF THE CONTRACT     :", address(market).balance);
}
```

Logs

```
[PASS] test_CanceledBidMatched() (gas: 376264)
Logs:
  ETH BALANCE OF THE CONTRACT     : 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.05s (1.65ms CPU time)
```

Recommendation

Ensure the `_clearMarket` function checks for canceled bids to prevent them from being settled.

Alternatively, consider removing canceled bids from storage in the `cancelBid` function.

4.1.3 The `_clearMarket` function is subject to denial of service due to an unbounded number of bids and asks

Description

The `_clearMarket` function is highly expensive in terms of gas costs. Due to the unbounded number of bids and asks that the market can have, the function is subject to denial of service. As a result, it won't be possible to clear the market for that hour.

This issue can occur through a malicious denial of service attack or simply because there are too many bids and asks.

Context

- EnergyBiddingMarket.sol#L304-L383

Impact

High. This issue will prevent the market from being cleared for a specific hour.

Likelihood

High. This issue has a high chance of occurring since there is no threshold in place on how many bids/asks a market can have per hour.

Proof of Concept

```
const { time } =
 ↪  require("@nomicfoundation/hardhat-toolbox/network-helpers");
const { ethers } = require('hardhat');

describe("Clear Market - Denial of Service", () => {
    const NUMBER_OF_BIDS = 700;
    const NUMBER_OF_ASKS = 700;

    let deployer, bidder, asker;

    before(async () => {
        // Get Hardhat signers
        [deployer, bidder, asker] = await ethers.getSigners();

        // Deploy market
        this.market = await
         ↪  ethers.deployContract("EnergyBiddingMarket", deployer);
        // Get target hour
        this.hour = await this.market.getCurrentHourTimestamp() +
         ↪  BigInt(3600);

        // Place bids
        for (let i = 0; i < NUMBER_OF_BIDS; ++i) {
            let tx = await
             ↪  this.market.connect(bidder).placeBid(this.hour, 1, {
             ↪  value: ethers.parseEther("0.000001") });
            await tx.wait();
        }

        // Skip time to place asks
        await time.increaseTo(this.hour);

        // Place asks
        for (let i = 0; i < NUMBER_OF_ASKS; ++i) {
            let tx = await this.market.connect(asker).placeAsk(1,
             ↪  asker.address);
            await tx.wait();
        }

        // Skip time to clear the market
        await time.increase(3600);
    });

    it("Should revert with out of gas", async () => {
        // This will revert due to out of gas
        await this.market.clearMarket(this.hour);
    }).timeout(120000);
});
```

> **Note:** The market was deployed without a proxy just to get a clearer error message on gas limit being exceeded.

Logs

```
Compiled 19 Solidity files successfully (evm target: paris).


  Clear Market - Denial of Service
```

```
    1) Should revert with out of gas


  0 passing (7s)
  1 failing

  1) Clear Market - Denial of Service
       Should revert with out of gas:
     ProviderError: Transaction ran out of gas
```

Instructions

1. Set up the `package.json` file by running the following command:

```
npm init -y
```

2. Install Hardhat as a dev dependency into your project:

```
npm i --save-dev hardhat
```

3. Initialize your Hardhat project inside the same directory and choose the "Create an empty `hardhat.config.js` " option:

```
npx hardhat init
```

4. Install the hardhat-foundry and the Hardhat toolbox plugins:

```
npm i --save-dev @nomicfoundation/hardhat-foundry
↪   @nomicfoundation/hardhat-toolbox
```

5. Place the following code into the `hardhat.config.js` file:

```
require("@nomicfoundation/hardhat-toolbox");
require("@nomicfoundation/hardhat-foundry");

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
  solidity: "0.8.28",
};
```

6. Add the `PoC.t.js` file inside the `test/` folder and place the provided PoC in it.
7. Run the test by running the following command:

```
npx hardhat test
```

Recommendation

Limit the number of bids and asks that the market can have by hour to prevent a denial of service from happening.

4.2 High Findings

4.2.1 Bids that weren't supposed to be matched are matched in the `_clearMarket` function, causing bidders to lose their funds

Description

The `_clearMarket` function is designed to settle bids and asks for a specific hour. However, under certain conditions, bids that should remain unmatched are instead matched, leading to bidders losing their funds.

Before diving into the issue, let's break down how the `_clearMarket` function operates:

1. Bids are sorted by price in descending order.
2. The clearing price of energy is determined by the price of the last matched bid.
3. The function iterates through the sorted bids, matching them with available asks.

The settlement process stops under two conditions:

- `bid.price < clearingPrice` : If the current bid's price is below the clearing price, it is deemed unmatched, and the process halts.
- `clearingPrice == 0` : If no bids were matched at all.

The issue arises when the first unmatched bid has a price equal to the clearing price. Since the stop condition only checks for bids priced below the clearing price, the bid bypasses the check and is improperly matched, even though no asks remain. As a result, that bidder loses access to their funds.

Example:

Consider the following bids:

- `BID 1` bidding for 10 kHw at a price of `0.1 ETH` .
- `BID 2` bidding for 10 kHw at a price of `0.1 ETH` .
- `BID 3` bidding for 10 kHw at a price of `0.1 ETH` .

Now, assume there are only 20 kWh of energy available from asks. Two bidders will have their bids matched, but since the clearing price is `0.1 ETH` , the third bid bypasses the stop condition (`bid.price < clearingPrice`) and is matched incorrectly, causing that bidder to lose funds.

Context

- EnergyBiddingMarket.sol#L320

Impact

High. This issue leads to bidders losing their funds due to unintended settlements.

Likelihood

Medium. The faulty condition allows bids to be improperly matched in several cases.

Proof of Concept

```solidity
function test_UnmatchedBidMatched() public {
    // Compute hour to bid
    uint256 hour = market.getCurrentHourTimestamp() + 3600;

    // Place bids
    vm.startPrank(BIDDER);
    market.placeBid{value: 1 ether}(hour, 100);
    market.placeBid{value: 1 ether}(hour, 100);
    vm.stopPrank();

    // Warp to the timestamp where askers begin to ask
    vm.warp(hour);

    // Asker places an ask to fully match the bid
    vm.prank(ASKER);
    market.placeAsk(100, ASKER);

    // Skip 1 hour so that we can clear the market and settle the orders
    skip(3600);

    // Clear the market
    market.clearMarket(hour);

    EnergyBiddingMarket.Bid[] memory bids = market.getBidsByHour(hour);

    // Assert that bid 1 is settled
    assertTrue(bids[0].settled);
    // Assert that bid 2 is settled
    assertTrue(bids[1].settled);
    // Assert that the asker's claimable balance corresponds only to the total eth amount of the first matched bid
    assertEq(market.claimableBalance(BIDDER), 0);
    // Assert that the bidder got no refund for the unmatched bid
    assertEq(market.claimableBalance(ASKER), 1 ether);
    // Assert that the market contract holds 2 ETH
    assertEq(address(market).balance, 2 ether);
    // Assert that the bidder cannot cancel the unmatched bid to
     ↪  recoup the funds

↪   vm.expectRevert(abi.encodeWithSelector(EnergyBiddingMarket__MarketAlreadyCle
↪   hour));
    vm.prank(BIDDER);
    market.cancelBid(hour, 1);

    // Logs bids settled flag
    console.log("BID 1 SETTLED  :", bids[0].settled);
    console.log("BID 2 SETTLED  :", bids[1].settled);
}
```

Logs

```
[PASS] test_UnmatchedBidMatched() (gas: 431059)
Logs:
  BID 1 SETTLED  : true
  BID 2 SETTLED  : true

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.65s
↪   (250.11µs CPU time)
```

Instructions

1. Add the following code to the `PoC.t.sol` file:

```
  // SPDX-License-Identifier: UNLICENSED
  pragma solidity ˆ0.8.13;

- import {EnergyBiddingMarket} from
↪   "../src/EnergyBiddingMarket.sol";
+ import {
+     EnergyBiddingMarket,
↪   EnergyBiddingMarket__MarketAlreadyClearedForThisHour
+ } from "../src/EnergyBiddingMarket.sol";

  contract PoC is Test {
      ...
  }
```

Recommendation

Change the `_clearMarket` function as follows to prevent such unintended settlements.

```
function _clearMarket(uint256 hour) internal
 ↪   isMarketNotCleared(hour) {
    ...
+   uint256 totalEnergyAvailable = totalAvailableEnergyByHour[hour];
+   uint256 totalMatchedEnergy;

    for (uint256 i = 0; i < totalBids; i++) {
        Bid storage bid = bidsByHour[hour][i];
-       if (bid.price < clearingPrice || clearingPrice == 0) {
+       if (bid.amount > totalEnergyAvailable - totalMatchedEnergy
 ↪   || clearingPrice == 0) {
            // if this consumes too much gas, change it to cancel
 ↪   bids
            for (uint k = i; k < totalBids; k++)
                claimableBalance[bidsByHour[hour][k].bidder] +=
                    bidsByHour[hour][k].amount *
                    bidsByHour[hour][k].price;
            break;
        }

        uint256 totalMatchedEnergyForBid = 0;

        for (uint256 j = fulfilledAsks; j < totalAsks; j++) {
            ...
        }

+       totalMatchedEnergy += bid.amount;

        ...
    }
    ...
}
```

4.2.2 Using `transfer` for native `ETH` withdrawals can prevent users from recouping their funds

Description

In the `claimBalance` function, `transfer` is used for native `ETH` withdrawals. The `transfer` and `send` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the `SLOAD` instruction.

The use of the deprecated `transfer` function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a `payable` function.
- The claimer smart contract does implement a `payable` fallback function which uses more than 2300 gas units.
- The claimer smart contract implements a `payable` fallback function that needs less than 2300 gas units but is called through a proxy, raising the call's gas usage above 2300.

Additionally, using more than 2300 gas might be mandatory for some multisig wallets.

Context

- EnergyBiddingMarket.sol#L234

Impact

High. This issue leads to users not being able to withdraw their funds from the contract.

Likelihood

Medium. This issue can occur in the aforementioned scenarios where the claimer is a smart contract.

Recommendation

Use `call` to prevent potential gas issues.

```
function claimBalance() external {
    uint256 balance = claimableBalance[msg.sender];
    if (balance == 0)
```

```
        revert EnergyBiddingMarket__NoClaimableBalance(msg.sender);
    claimableBalance[msg.sender] = 0;
-   payable(msg.sender).transfer(balance);
+   (bool success,) = msg.sender.call{value: balance}("");
+   require(success, "ETH transfer failed");
}
```

4.4 Low Findings

4.4.1 Truncation in `placeBid` function is retained by the contract

Description

When placing a bid, excess Ether sent due to integer division truncation is retained by the contract and not refunded, leading to permanent loss of funds for the bidder.

In the `placeBid` function, the `price` is calculated as `msg.value / amount` using integer division. If `msg.value` is not perfectly divisible by `amount`, the remainder (excess Ether) is not returned to the bidder. This results in silent loss of funds.

Context

- EnergyBiddingMarket.sol#L134-L142

Impact

Low. Bidders may unintentionally lose small amounts of Ether (due to truncation) when placing bids with values not perfectly divisible by the bid `amount`.

Likelihood

Medium. This issue arises whenever bidders send values not aligned with `amount * price`. Human error or misconfigured frontends could trigger this frequently.

Proof of Concept

```
function test_PlaceBidResiduals() public {
    // Calculate current hour using block timestamp
    uint256 currentHour = (block.timestamp / 3600) * 3600;
    uint256 hour = currentHour + 3600;

    uint256 energyAmount = 1779; // 1779 kWh
    uint256 ethAmount = 999999900000022007; //
     ↪   0.999999900000022007 ETH

    // Calculate expected truncated price
    uint256 expectedPrice = ethAmount / energyAmount;
    console.log("Expected price      :", expectedPrice);

    // Execute bid placement
    vm.deal(BIDDER, ethAmount);
    vm.prank(BIDDER);
    market.placeBid{value: ethAmount}(hour, energyAmount);

    // Get stored bid details
    (,,, uint256 actualAmount, uint256 actualPrice) =
↪   market.bidsByHour(hour, 0);

    // Verify price calculation
    assertEq(actualPrice, expectedPrice, "Incorrect price
     ↪   calculation");
    console.log("Actual price        :", actualPrice);

    // Calculate expected residual
    uint256 expectedResidual = ethAmount - (expectedPrice *
     ↪   energyAmount);
    assertGt(expectedResidual, 0, "No residual amount");
    console.log("Expected residual   :", expectedResidual);

    // Verify full amount retained (including residuals)
    assertEq(address(market).balance, ethAmount, "Incorrect ETH
     ↪   balance");
    console.log("Total residual      :", ethAmount - (actualPrice *
↪   actualAmount));
}
```

Logs

```
[PASS] test_PlaceBidResiduals() (gas: 134626)
Logs:
  Expected price      : 5621134907251277
  Actual price        : 5621134907251277
  Expected residual   : 224
  Total residual      : 224

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.25s
↪   (521.49µs CPU time)
```

Recommendation

Enforce exact payment or refund residuals:

```
function placeBid(uint256 hour, uint256 amount)
    external
    payable
    assertExactHour(hour)
{
    if (amount == 0) revert
 ↪  EnergyBiddingMarket__AmountCannotBeZero();

    uint256 price = msg.value / amount;
+   uint256 totalCost = price * amount;

+   // Revert if overpaid (strict equality check)
+   if (msg.value != totalCost) {
+       revert EnergyBiddingMarket__ExactValueRequired(totalCost);
+   }

    _placeBid(hour, amount, price);
}
```

OR

Refund excess Ether to the bidder in the `placeBid` function:

```
function placeBid(uint256 hour, uint256 amount)
    external
    payable
    assertExactHour(hour)
{
    if (amount == 0) revert
 ↪  EnergyBiddingMarket__AmountCannotBeZero();

    uint256 price = msg.value / amount;
+   uint256 totalCost = price * amount;
+   uint256 excess = msg.value - totalCost;

    _placeBid(hour, amount, price);

    // Refund excess Ether
+   if (excess > 0) {
+       (bool sucess, ) = msg.sender.call{value: excess}("");
+       require(sucess, "It tranfer failed");
+   }
}
```

OR

```
function placeBid(
    uint256 hour,
    uint256 amount,
+   uint256 price
) external payable assertExactHour(hour) {
    if (amount == 0) revert EnergyBiddingMarket__AmountCannotBeZero();
+   require(msg.value == amount * price, "invalid eth amount");

-   uint256 price = msg.value / amount;
    _placeBid(hour, amount, price);
}
```

4.4.2 Bulk bid residuals in multi-hour bidding

Description

When placing multiple bids via `placeMultipleBids(uint256,uint256,uint256)`, residual Ether accumulates due to inexact division across the total bid hours, resulting in permanent fund loss.

The `placeMultipleBids(uint256,uint256,uint256)` function calculates a uniform `price` as `msg.value / (amount * hours)`. If `msg.value` is not perfectly divisible by the total energy-hours (`amount * hours`), the remainder is retained by the contract. Unlike single-bid truncation, this issue arises from bulk allocation across multiple hours, leading to larger residuals.

Context

- EnergyBiddingMarket.sol#193-L207

Impact

Low. Residuals scale with the number of hours (e.g., 24-hour bids compound truncation losses) and the excess Ether from bulk calculations is permanently locked in the contract.

Likelihood

Medium. Likely to occur in multi-hour bids due to frequent misalignment between `msg.value` and `amount * hours * price`.

Proof of Concept

```
function test_PlaceMultipleBidsResiduals() public {
    // 1. Limit parameters to safe ranges
    uint256 numHours = 18; // 18 hours
    uint256 energyAmount = 9;

    // 2. Constrain ethAmount to valid range
    uint256 ethAmount = 9999838000000018636; // 9999838000000018636 wei

    // 3. Calculate time parameters
    uint256 currentHour = (block.timestamp / 3600) * 3600;
    uint256 beginHour = currentHour + 3600;
    uint256 endHour = beginHour + (numHours * 3600);
    // 4. Execute multiple bids
    vm.deal(BIDDER, ethAmount);
    vm.startPrank(BIDDER);
    market.placeMultipleBids{value: ethAmount}(beginHour, endHour, energyAmount);
    vm.stopPrank();

    // 5. Verify residual calculations
    uint256 totalUsed;
    for (uint256 hour = beginHour; hour < endHour; hour += 3600) {
        (,,, uint256 amount, uint256 price) = market.bidsByHour(hour, 0);
        totalUsed += amount * price;
    }

    console.log("Total residual:", ethAmount - totalUsed);
    assertEq(address(market).balance, ethAmount, "Contract should retain full amount including residuals");

    assertGt(ethAmount - totalUsed, 0, "No residual amount");
}
```

Logs

```
[PASS] test_PlaceMultipleBidsResiduals() (gas: 1834618)
Logs:
  Total residual: 16

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.97s (6.63ms CPU time)
```

Recommendation

Refund bulk residuals after placing all bids:

```
function placeMultipleBids(uint256 beginHour, uint256 endHour, uint256 amount)
    external
    payable
    assertExactHour(beginHour)
    assertExactHour(endHour)
```

```
{
     if (amount == 0) revert
 ↪  EnergyBiddingMarket__AmountCannotBeZero();

     if (beginHour + 3600 > endHour) {
         revert EnergyBiddingMarket__WrongHoursProvided(beginHour,
 ↪  endHour);
     }

     // Calculate and refund residual
+    uint256 totalEnergy = ((amount * (endHour - beginHour)) /
 ↪  3600);
+    uint256 price = msg.value / totalEnergy;
+    uint256 totalCost = price * totalEnergy;
+    uint256 excess = msg.value - totalCost;

-    uint256 price = msg.value / ((amount * (endHour - beginHour)) /
 ↪  3600);
     for (uint256 i = beginHour; i < endHour; i += 3600) {
         _placeBid(i, amount, price);
     }

+    if (excess > 0) {
+        (bool success, ) = msg.sender.call{value: excess}("");
+        require(success, "ETH transfer failed");
+    }
}
```

OR

Refund excess Ether to the bidder in the `placeMultipleBids` function:

```
function placeMultipleBids(uint256 beginHour, uint256 endHour,
 ↪  uint256 amount)
    external
    payable
    assertExactHour(beginHour)
    assertExactHour(endHour)
{
     if (amount == 0) revert
 ↪  EnergyBiddingMarket__AmountCannotBeZero();
```

```
     if (beginHour + 3600 > endHour) {
         revert EnergyBiddingMarket__WrongHoursProvided(beginHour,
 ↪  endHour);
     }

+    uint256 totalEnergy = ((amount * (endHour - beginHour)) /
 ↪  3600);
-    uint256 price = msg.value / ((amount * (endHour - beginHour)) /
 ↪  3600);
+    uint256 price = msg.value / totalEnergy;
+    uint256 totalCost = price * totalEnergy;

+    // Revert if overpaid (strict equality check)
+    if (msg.value != totalCost) {
+        revert EnergyBiddingMarket__ExactValueRequired(totalCost);
 ↪
+    }

     for (uint256 i = beginHour; i < endHour; i += 3600) {
         _placeBid(i, amount, price);
     }
}
```

OR

```
- function placeMultipleBids(uint256 beginHour, uint256 endHour,
 ↪  uint256 amount)
+ function placeMultipleBids(uint256 beginHour, uint256 endHour,
 ↪  uint256 amount, uint256 price)
    external
    payable
    assertExactHour(beginHour)
    assertExactHour(endHour)
{
    if (amount == 0) revert
 ↪  EnergyBiddingMarket__AmountCannotBeZero();

    if (beginHour + 3600 > endHour) {
        revert EnergyBiddingMarket__WrongHoursProvided(beginHour,
 ↪  endHour);
    }
+   uint256 totalEnergy = ((amount * (endHour - beginHour)) / 3600);
+   require(msg.value == totalEnergy * price, "invalid eth amount");

-   uint256 price = msg.value / ((amount * (endHour - beginHour)) /
 ↪  3600);
    for (uint256 i = beginHour; i < endHour; i += 3600) {
        _placeBid(i, amount, price);
    }
}
```

4.4.3 Array-based bulk bid residuals

Description

When placing multiple bids via `placeMultipleBids(uint256[], ...)`, residual Ether is retained due to truncation in price calculation. The function computes `price = msg.value / (amount * bidsAmount)` using integer division, and any remainder is locked in the contract, causing permanent fund loss for bidders.

Context

- EnergyBiddingMarket.sol#213-L225

Impact

Low. Residuals scale with the number of hours in the `biddingHours` array. Bidders lose small amounts of Ether proportional to the truncation error multiplied by the array length.

Likelihood

Medium. Likely to occur when `msg.value` is not a multiple of `amount * bidsAmount`, which is common in real-world scenarios with variable pricing.

Proof of Concept

```
function test_ArrayBulkBidResiduals() public {
    // 1. Define parameters
    uint256[] memory hoursArray = new uint256[](17); // 17-hour bids
    uint256 currentHour = (block.timestamp / 3600) * 3600;
    for (uint256 i = 0; i < 17; i++) {
        hoursArray[i] = currentHour + 3600 * (i + 1); // Future
↪   hours
    }

    uint256 energyAmount = 13; // 13 kWh per bid
    uint256 ethAmount = 9999999000000022007; // 9999999000000022007
     ↪   wei

    // 2. Place bids
    vm.deal(BIDDER, ethAmount);
    vm.prank(BIDDER);
    market.placeMultipleBids{value: ethAmount}(hoursArray,
↪   energyAmount);

```

```
    // 3. Calculate residuals
    uint256 totalUsed;
    for (uint256 i = 0; i < hoursArray.length; i++) {
        (,,, uint256 amount, uint256 price) = market.bidsByHour(hoursArray[i], 0);
        totalUsed += amount * price;
    }

    console.log("Total residual:", ethAmount - totalUsed);
    assertGt(ethAmount - totalUsed, 0, "No residual retained");
    assertEq(address(market).balance, ethAmount, "Full amount locked");
}
```

Logs

```
[PASS] test_ArrayBulkBidResiduals() (gas: 1749998)
Logs:
  Total residual: 163

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.05s (3.28ms CPU time)
```

Recommendation

Refund bulk residuals after placing all bids:

```
function placeMultipleBids(
    uint256[] memory biddingHours,
    uint256 amount
) external payable {
    if (amount == 0) revert EnergyBiddingMarket__AmountCannotBeZero();

    uint256 bidsAmount = biddingHours.length;
+   uint256 totalEnergy = (amount * bidsAmount);
+   uint256 price = msg.value / totalEnergy;
+   uint256 totalCost = price * totalEnergy;
+   uint256 excess = msg.value - totalCost;

-   uint256 price = msg.value / (amount * bidsAmount);
```

```
    for (uint256 i = 0; i < bidsAmount; i++) {
        _placeBid(biddingHours[i], amount, price);
    }

+   if (excess > 0) {
+       (bool success, ) = msg.sender.call{value: excess}("");
+       require(success, "ETH transfer failed");
+   }
}
```

OR

Refund excess Ether to the bidder in the `placeMultipleBids` function:

```
function placeMultipleBids(
    uint256[] memory biddingHours,
    uint256 amount
) external payable {
    if (amount == 0) revert
 ↪  EnergyBiddingMarket__AmountCannotBeZero();

    uint256 bidsAmount = biddingHours.length;
+   uint256 totalEnergy = (amount * bidsAmount);
+   uint256 price = msg.value / totalEnergy;
+   uint256 totalCost = price * totalEnergy;

+   if (msg.value != totalCost) {
+       revert EnergyBiddingMarket__ExactValueRequired(totalCost);
+   }

-   uint256 price = msg.value / (amount * bidsAmount);
    for (uint256 i = 0; i < bidsAmount; i++) {
        _placeBid(biddingHours[i], amount, price);
    }
}
```

OR

```
function placeMultipleBids(
    uint256[] memory biddingHours,
    uint256 amount,
+   uint256 price
) external payable {
    if (amount == 0) revert EnergyBiddingMarket__AmountCannotBeZero();

    uint256 bidsAmount = biddingHours.length;
+   require(msg.value == amount * bidsAmount * price, "invalid eth amount");

-   uint256 price = msg.value / (amount * bidsAmount);
    for (uint256 i = 0; i < bidsAmount; i++) {
        _placeBid(biddingHours[i], amount, price);
    }
}
```

4.4.4 Missing `_disableInitializers` in `EnergyBiddingMarket` ’s constructor

Description

The `EnergyBiddingMarket` contract is designed to be upgradable, using the Universal Upgradeable Proxy Standard (UUPS). However, the contract does not diable the initializer in its constructor, leaving it vulnerable to potential initialization attacks.

Context

- EnergyBiddingMarket.sol

Recommendation

Disable the initializers in the constructor:

```
...
contract EnergyBiddingMarket is Initializable, UUPSUpgradeable,
 ↪  OwnableUpgradeable {
    ...
+   /// @custom:oz-upgrades-unsafe-allow constructor
+   constructor() {
+       _disableInitializers();
+   }
    ...
}
```

4.4.5 There is no way to undo the whitelisting of a seller

Description

In the current implementation of `EnergyBiddingMarket` contract, there is no mechanism in place to undo the whitelisting of a seller. A seller's private key might be leaked and, consequently, expose the protocol to interact with a malicious actor.

Context

- EnergyBiddingMarket.sol

Recommendation

Add the possibility to undo the whitelisting of a seller.

```
- function whitelistSeller(address seller) external onlyOwner {
+ function whitelistSeller(address seller, bool enable) external
 ↪  onlyOwner {
-     s_whitelistedSellers[seller] = true;
+     s_whitelistedSellers[seller] = enable;
  }
```

4.5 Informational Findings

4.5.1 Unused `whitelistedSeller` modifier

Description

The `whitelistedSeller` modifier is defined but never applied, rendering the seller whitelisting mechanism non-functional and creating dead code.

The modifier `whitelistedSeller` checks if the caller is whitelisted but is not used in the `placeAsk` function (where it was intended). This leaves the whitelist feature unimplemented, allowing unauthorized sellers to participate.

Context

- EnergyBiddingMarket.sol#268-L270

Recommendation

Activate the whitelist check by uncommenting the modifier in `placeAsk` or remove the modifier.

4.5.2 Unnecessary repeated calls to `assertExactHour` modifier when placing bids

Description

The external functions used for bids placement redundantly apply the `assertExactHour` modifier, even though the internal `_placeBid` function already enforces this check. This redundancy leads to unnecessary gas consumption and duplicated logic.

Context

- EnergyBiddingMarket.sol#278-L294

Recommendation

To address the redundancy, centralize the `assertExactHour` modifier within the `_placeBid` function, removing it from the parent functions that call `_placeBid`. This approach ensures each hour is checked exactly once, optimizing gas usage.

4.5.3 Unnecessary `getClearingPrice` function

Description

The `getClearingPrice` function is redundant because the `clearingPricePerHour` mapping is declared as `public`, which auto-generates a getter function with identical functionality.

In Solidity, `public` mappings automatically generate a getter function in the form `clearingPricePerHour(uint256)`. The custom `getClearingPrice` function duplicates this behavior, offering no added value while increasing code redundancy.

Context

- EnergyBiddingMarket.sol#544-L547

Recommendation

Remove the redundant `getClearingPrice` so it could safe deployment cost and use the auto-generated `clearingPricePerHour` getter instead.

4.5.4 Lack of `address(0)` sanity check in `whitelistSeller` function

Description

The `whitelistSeller` function allows the owner to whitelist the zero address (`address(0)`), violating basic input validation and risking system integrity.

The `whitelistSeller` function lacks a check to prevent whitelisting address(0). This could lead to:

- Invalid "ghost" sellers in the system.
- Potential logical errors in features relying on valid seller addresses.

Context

- EnergyBiddingMarket.sol#268-L270

Recommendation

Add a zero-address check in whitelistSeller:

```
function whitelistSeller(address seller) external onlyOwner {
+   require(seller != address(0), "Invalid seller address");
    s_whitelistedSellers[seller] = true;
}
```

This prevents invalid address entries and enforces data cleanliness.

4.5.5 Lack of event emission for state change in `whitelistSeller` function

Description

The `whitelistSeller` function modifies critical state (`s_whitelistedSellers`) but does not emit an event, violating transparency standards and hindering off-chain monitoring.

Events are essential for tracking state changes in decentralized systems. The absence of an event when whitelisting sellers makes it impossible to:

- Audit permission changes retroactively.
- Trigger off-chain actions (e.g., notifications, indexing).
- Prove historical whitelist status without direct chain inspection.

Context

- EnergyBiddingMarket.sol#268-L270

Recommendation

Emit an event when whitelisting sellers:

```
event SellerWhitelisted(address indexed seller);

function whitelistSeller(address seller) external onlyOwner {
    s_whitelistedSellers[seller] = true;
+   emit SellerWhitelisted(seller);
}
```

This ensures auditable and reactive tracking of permission changes.

4.5.6 Unnecessary `balanceOf` function

Description

The `balanceOf` function is redundant because the `claimableBalance` mapping is already declared as `public`, which auto-generates a getter function with the same functionality.

In Solidity, `public` mappings automatically generate a getter function named after the mapping (e.g., `claimableBalance(address)`). The custom `balanceOf` function duplicates this behavior unnecessarily, adding no value while increasing code redundancy.

Context

- EnergyBiddingMarket.sol#300-L383

Recommendation

Remove the redundant `balanceOf` function so it could safe deployment cost and use the auto-generated `claimableBalance` getter instead.

4.5.7 Missing bid existence checks in cancellation functions

Description

The function `cancelBid` do not verify if the specified bid index exists for the given hour. This leads to misleading error messages when interacting with non-existent bids.

When canceling a bid, the contract checks ownership and market status but does not validate if the provided `index` is within the valid range of bids for the hour.

Context

- EnergyBiddingMarket.sol#258-L270

Recommendation

Add explicit index validation to cancellation functions:

```
function cancelBid(uint256 hour, uint256 index) external
 ↪  isMarketNotCleared(hour) {
+   require(index < totalBidsByHour[hour], "Bid does not exist");
    // ... existing checks ...
}
```

Or, introduce a custom error:

```
+ error EnergyBiddingMarket__BidDoesNotExist(uint256 hour, uint256
 ↪  index);

function cancelBid(...) {
    if (index >= totalBidsByHour[hour]) {
+       revert EnergyBiddingMarket__BidDoesNotExist(hour, index);
    }
    // ...
}
```

This ensures clear, accurate error reporting for invalid indices.

4.5.8 Inefficient `_clearMarket` function

Description

The internal `_clearMarket` function is implemented inefficiently gas-wise.

Context

- EnergyBiddingMarket.sol#L304-L383

Recommendation

Follow gas optimization best practices, such as:

- If there is no bids nor asks, proceed with the bidders' refund immediately.
- Implement an already well-known, optimized sorting algorithm.
- Use memory over storage whenever possible.
- Use while loops over for loops.
- Use `unchecked` code blocks whenever possible.

For more information, please visit The RareSkills Book of Solidity Gas Optimization: 80+ Tips.