OperatorUtilizationHeap::Duplicating last operator in the heap while removing an operator

Summary

Due to miscalculation in OperatorUtilizationHeap::_remove() a duplicate of last operator is made while removing an operator.

Vulnerability Detail

First run this test in OperatorUtilizationHeap.t.sol contract:

```
function test_removing() public {
        OperatorUtilizationHeap.Data memory heap = OperatorUtilizationHeap.initialize(5);
        assertTrue(heap.isEmpty());
        assertEq(heap.count, 0);
        heap.insert(OperatorUtilizationHeap.Operator({id: 1, utilization: 15}));
        heap.insert(OperatorUtilizationHeap.Operator({id: 2, utilization: 5}));
        heap.insert(OperatorUtilizationHeap.Operator({id: 3, utilization: 10}));
        console.log("count is:", heap.count);

        assertEq(heap.operators[1].id, 2);
        assertEq(heap.operators[1].utilization, 5);
        assertEq(heap.operators[2].id, 1);
        assertEq(heap.operators[2].utilization, 15);
        assertEq(heap.operators[3].id, 3);

        assertEq(heap.operators[3].utilization, 10);

        heap.remove(2); // removing operator of index 2

        assertEq(heap.operators[1].id, 2);
        assertEq(heap.operators[1].utilization, 5);
        assertEq(heap.operators[2].id, 3);   // 2nd operator removed & 3rd index's operator came here
        assertEq(heap.operators[2].utilization, 10);
        assertEq(heap.operators[3].id, 3);  // the 3rd operator still in index 3
        assertEq(heap.operators[3].utilization, 10);

        console.log("count after removing is:", heap.count);
}
```

If we run this test it will succeed:

```
Ran 1 test for test/OperatorUtilizationHeap.t.sol:OperatorUtilizationHeapTest
[PASS] test_removing() (gas: 11482)
Logs:
  count is: 3
  count after removing is: 2

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 573.96µs

Ran 1 test suite in 573.96µs: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Lets dive deep into removing the 2nd index operator. The `OperatorUtilizationHeap::remove()` looks like this:

```
    /// @notice Removes an operator from the heap.
    /// @param self The heap.
    /// @param index The index of the operator to remove.
    function remove(Data memory self, uint8 index) internal pure {
        if (index < ROOT_INDEX || index > self.count) revert INVALID_INDEX();
        self._remove(index);
        self._bubbleUp(index);
        self._bubbleDown(index);
    }
```

In test we called this remove() with index = 2. At this point heap.count is 3 & there is total 3 operator in operators[ ]. So, 1st it will call the _remove(). The function looks like this:

```
    function _remove(Data memory self, uint8 i) internal pure {
        self.operators[i] = self.operators[self.count--];
    }
```

As i = 2 here operators[ ]'s 2nd index is assigned with self.count i.e 3, then it is substracted by 1. Now 3rd index's operator is in 2nd index. If we see the test snippet we can see:

```
        assertEq(heap.operators[2].id, 3);
        assertEq(heap.operators[2].utilization, 10);
```

So, the index 2 successfully substituted by the operator in 3rd index. But the 3rd index's operator i.e the last operator is still in last index i.e 3rd index.

```
        assertEq(heap.operators[3].id, 3);
        assertEq(heap.operators[3].utilization, 10);
```

A duplicate of 3rd operator was made. Now, _bubbleUp() was called with index 2. As index = 2 the parentIndex inside it will be 1:

```
uint8 parentIndex = i / 2;
```

But the condition on next line: if (_isOnMinLevel(i)) { evaluated to false for i = 2, so the else part will be executed, in else block the condition:

```
if (_hasParent(i) && self.operators[i].utilization < self.operators[parentIndex].utilization) {
```

will evaluate to false because for index 2 the _hasParent(i) is true but as 2nd operator's utilization is more than 1st operator's utilization. As this condition evaluated to false the next else part: self._bubbleUpMax(i) will execute and here this also does nothing because in _bubbleUpMax() the if (_hasGrandparent(i)) { condition evaluated to false. As we can till now nothing changed in call flow of self._bubbleUp(index);. Next, self._bubbleDown(index); is called with index 2, here if (_isOnMinLevel(i)) { condition evaluate to false so the else block will execute, in else block _bubbleDownMax() is called with the index 2, in this function if (self._hasChildren(i)) { condition will evaluate to false so this _bubbleDownMax() also does nothing for index 2.

As we saw the 2nd operator [from index 2] was replaced by the 3rd operator but the 3rd operator was not removed from 3rd index, so resulted duplication. The heap.count in decreased so it also creates a mismatch.

Impact

Allowing the same operator to run Ethereum validators twice risks double signing, centralizes power, increases economic concentration, and undermines protocol integrity, jeopardizing network stability and decentralization principles.

Code Snippet

- https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/utils/OperatorUtilizationHeap.sol#L388-L390 (https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/utils/OperatorUtilizationHeap.sol#L388-L390)
- https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/utils/OperatorUtilizationHeap.sol#L94-L100 (https://github.com/sherlock-audit/2024-02-rio-network-core-protocol/blob/main/rio-sherlock-audit/contracts/utils/OperatorUtilizationHeap.sol#L94-L100)

Tool used

Manual Review

Recommendation

Remove the last operator from operators[ ] after swapping.