Revolution Protocol

Scope

The code under review can be found within the C4 Revolution Protocol repository.

Summary

In this contest, I discovered 1 High-severity issue and 3 Medium-severity issues with my Gas Optimizations report being selected for the final public report.

Findings

ID	Issues	Severity
H-01	ETH used to calculate ERC20 tokens for creatorsAddress is permanently locked in ERC20TokenEmitter.sol	High
M-01	positionMapping for last element in heap is not updated when extracting max element	Medium
M-02	Last bidder can win auction by force sending ETH through selfdestruct() and not placing a new bid	Medium
M-03	Last element can be overwritten and removed from existence in max heap tree	Medium
N-01	Require check in getArtPieceById() function should use < instead of <=	Non-Critical

ID	Issues	Severity
N-02	Redundant check in _mintTo() function can be removed	Non-Critical
N-03	Incorrect comment for mapping heap	Non-Critical
N-04	No need to initialize unsigned integer to 0	Non-Critical
N-05	Remove redundant check from TokenEmitterRewards.sol	Non-Critical
L-01	PUSH0 opcode is not supported on BASE AND Optimism chains	Low
L-02	Prefer prioritizing the left hand side of the tree than right in binary tree	Low
L-03	createPiece() can be spammed on cheap fee chains like Base and Optimism	Low
L-04	Return value of extractMax() not checked in function dropTopVotedPiece() in CultureIndex	Low
L-05	1.75% of ETH provided to buyToken() can be refunded by users to themselves when buying the ERC20 tokens	Low
L-06	Missing field and check for audio metadata type length to be greater than 0	Low
L-07	ERC20Votes liquidity fragmentation and art theft can occur if contracts are deployed across three different chains	Low
L-08	Use >= instead of > when using minVoteWeight as criteria to start voting	Low
L-09	Code does not follow spec and breaks soft invariant in pause/unpause functionalities	Low
R-01	Expose burn() function with access control in NontransferableERC20Votes.sol for any future supply mechanism purposes	Recommendation
R-02	Consider removing the `_minCreatorRateBps > minCreatorRateBps` check	Recommendation

Gas Optimizations

ID	Optimization
G-01	`MAX_NUM_CREATORS` check is not required when minting in VerbsToken.sol contract

ID	Optimization
G-02	Mappings not used externally/internally can be marked private
G-03	No need to initialize variable `size` to 0
G-04	Use `left + 1` to calculate value of `right` in `maxHeapify()` to save gas
G-05	Place size check at the start of maxHeapify() to save gas on returning case
G-06	Cache `parent(current)` in insert() function to save gas
G-07	else-if block in function updateValue() can be removed since newValue can never be less than oldValue
G-08	`size > 0` check not required in function getMax()
G-09	Cache `msgValueRemaining - toPayTreasury` in buyToken() to save gas
G-10	`creatorsAddress != address(0)` check not required in buyToken()
G-11	Cache return value of _calculateTokenWeight() function to prevent SLOAD
G-12	Cache `erc20VotingToken.totalSupply()` to save gas
G-13	Unnecessary for loop can be removed by shifting its statements into an existing for loop
G-14	Return memory variable `pieceId` instead of storage variable `newPiece.pieceId` to save gas
G-15	Calculate `creatorsShare` before `auctioneerPayment` in buyToken() to prevent unnecessary SUB operation
G-16	Remove `msgValue < computeTotalReward(msgValue` check from TokenEmitterRewards.sol contract
G-17	Optimize `computeTotalReward()` and `computePurchaseRewards` into one function to save gas
G-18	Calculation in computeTotalReward() can be simplified to save gas
G-19	Negating twice in require check is not required in _vote() function

Findings

[H-01] ETH used to calculate ERC20 tokens for creatorsAddress is permanently locked in ERC20TokenEmitter.sol

Impact

In buyToken() function, after calculating the rewards for the ProtocolRewards contract, treasury and creatorDirectPayment, the remaining ether used to calculate the ERC20 tokens to be minted is permanently locked in the ERC20TokenEmitter.sol contract. The greater the creatorRateBps and smaller the entropyRateBps, the higher is the amount of ETH permanently locked in the contract.

Greater the creatorRateBps = higher is the cut for creatorsAddress compared to treasury Smaller the entropyRateBps = lower is the ETH cut and higher is the ETH (which is locked) used to calculate ERC20 cut for creatorsAddress

Proof of Concept

Here is the whole process:

1. Currently we know that when we call buyToken(), the ETH is split into these parts:

- 2.5% of msg.value to the ProtocolRewards contract
- 10000 - creatorRateBps of the remaining amount to treasury
- entropyRateBps of the remaining amount to creatorsAddress in ETH directly
- Remaining ETH is used to calculate ERC20 tokens to mint to creatorsAddress. (This is the ETH that is permanently locked)

2. Let's walk through an example. Let's say we want to buy 10 ETH worth of tokens through function buyToken().

creatorRateBps = 5000 entropyRateBps = 5000 Initial value = 10 ETH = 10 * (10 ^ 18)

Protocol Rewards contract

Protocol Rewards split = 2.5 % of 10 ETH = 2.5 * (10 ^ 17) ETH

Remaining ETH = Initial value - Protocol Rewards split = 9.75 * (10 ^ 18) ETH

Treasury

Treasury split = (Remaining ETH * (10000 - 5000)) / 10000 = 4.875 * (10 ^ 18) ETH

Remaining ETH = Remaining ETH - Treasury split = 4.875 * (10 ^ 18) ETH

Creator Direct Payment

Creator direct payment in ETH = (Remaining ETH * 5000) / 10000 = 2.4375 * (10 ^ 18) ETH

Remaining ETH = Remaining ETH - Creator direct payment = 2.4375 * (10 ^ 18) ETH

This final amount of remaining ETH is used to calculate ERC20 tokens to mint to the creatorsAddress but after the calculation is done, the ETH is never withdrawn or able to be withdrawn, leaving it permanently

locked in the ERC20TokenEmitter.sol contract.

In this case, the greater the creatorRateBps and smaller the entropyRateBps, the higher is the amount of ETH permanently locked in the contract.

Note: All calculations done above are based on 1, 2, 3, 4 respective equations and rates.

Tools Used

Manual Review

Recommended Mitigation Steps

Implement a withdrawal function to withdraw this locked ETH, which could be used for other protocol related purposes.

[M-01] positionMapping for last element in heap is not updated when extracting max element

Impact

During the extraction of the max element through the function extractMax(), the `positionMapping` for the last element in the heap tree is not updated when the last element is equal to its parent.

These are the following impacts:

1. MaxHeap does not function as intended and breaks its expected functionality due to element being incorrectly indexed in the heap.
2. When a new element is inserted to that index, the incorrectly indexed element accesses the new element's itemId during value updates
3. Downward heapifying will work incorrectly leading to the parent having a value smaller than its child. Thus, further breaking not only the maxHeap tree but also the binary tree spec.
4. Error in this heapifying will lead to the incorrectly indexed element being extracted.

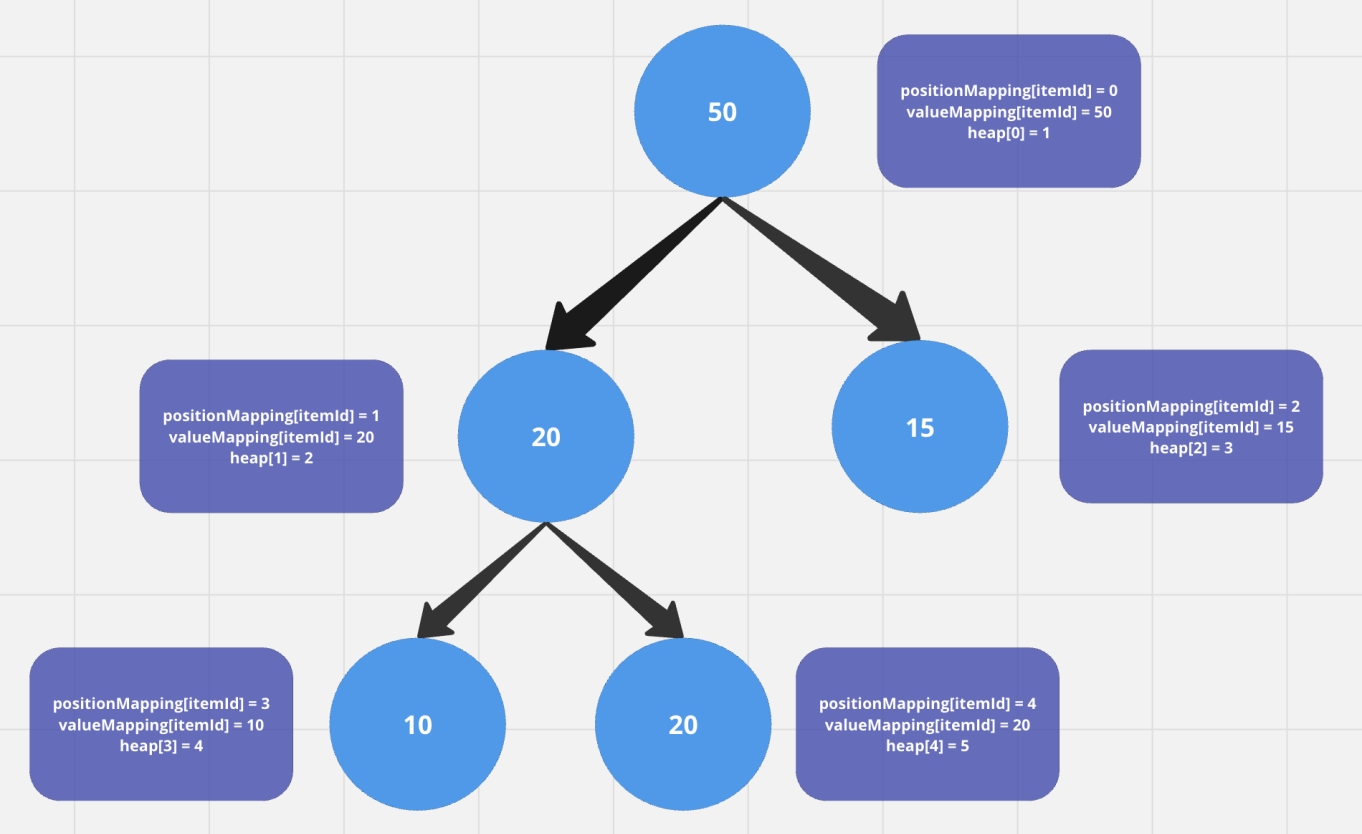
Proof of Concept

Here is the whole process:

1. Let's assume this is the current state of the maxHeap tree.

- The values have been made small for ease of demonstration of the issue
- The itemIds being used are 1,2,3,4,5 with values being 50,20,15,10,20 respectively.
- The current indexes of the items are 0,1,2,3,4 (since size starts from 0)
- The current size of the tree is 5.
- Over here, note that **itemId 2 is the parent of itemId 5 and both have the same values 20**.

Link to image



2. Now let's say the top element with itemId = 1 and value = 50 is extracted using the extractMax() function. The following occurs:

- On Line 174, we store the itemId at the root into the variable popped
- On Line 175, we replace the itemId at the root (effectively erasing it) with the last element in the heap tree i.e. itemId 5.
- On Line 175, size is decremented from 5 to 4 as expected.
- Following this on Line 178, we maxHeapify the current element at the root i.e. itemId 5 which was just set on Line 174.

```
File: MaxHeap.sol
171:     function extractMax() external onlyAdmin returns (uint256,
uint256) {
172:         require(size > 0, "Heap is empty");
173:
174:         uint256 popped = heap[0];
175:         heap[0] = heap[--size];
176:
177:
178:         maxHeapify(0);
179:
180:         return (popped, valueMapping[popped]);
181:     }
```

3. During the maxHeapify() internal function call, the following occurs:

- On Lines 100-101, left and right are the indexes 1 and 2 (since pos = 0)

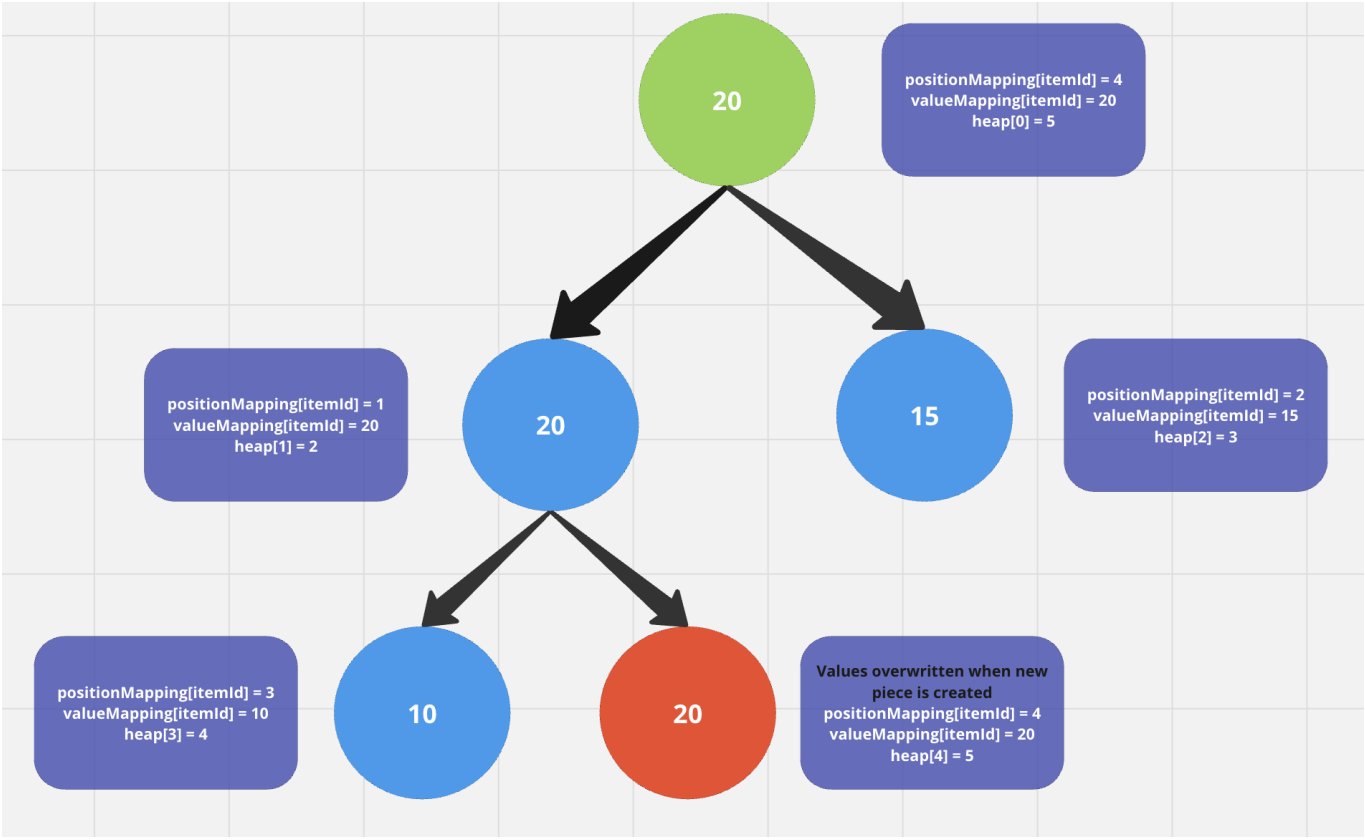
- On Lines 103-105, the values of the respective itemIds 5,2,3 are extracted as 20,20,15 respectively.
- On Line 107, we pass the check since 0 is not >= 2 (since size/2 = 4/2 = 2)
- On Line 109, we do not enter the if block since in the tree now, itemId 5 at the root has value 20 which is neither less than itemId 2 with value 20 nor itemId 3 with value 15. Thus, the function returns back to extractMax() and the call ends.

```
File: MaxHeap.sol
099:     function maxHeapify(uint256 pos) internal {
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = 2 * pos + 2;
102:
103:         uint256 posValue = valueMapping[heap[pos]];
104:         uint256 leftValue = valueMapping[heap[left]];
105:         uint256 rightValue = valueMapping[heap[right]];
106:
107:         if (pos >= (size / 2) && pos <= size) return;
108:
109:         if (posValue < leftValue || posValue < rightValue) {
110:
111:             if (leftValue > rightValue) {
112:                 swap(pos, left);
113:                 maxHeapify(left);
114:             } else {
115:                 swap(pos, right);
116:                 maxHeapify(right);
117:             }
118:         }
119:     }
```

4. On observing the new state of the tree, we can see the following:

- ItemId 5 with value 20 has been temporarily removed from index 4 in the tree until a new item is inserted.
- ItemId 5 with value 20 is now the root of the tree.
- If we look at the details of the root node, we can see that although the heap and valueMapping mappings have been updated correctly, **the positionMapping still points to index 4 for itemId 5**.
- This issue originates because during the maxHeapify() call, the itemId 5 is not less than either of it's child nodes as the condition in maxHeapify() demands on Line 109 above.
- This is the first impact on the maxHeap data structure since it does not index as expected.

Link to image



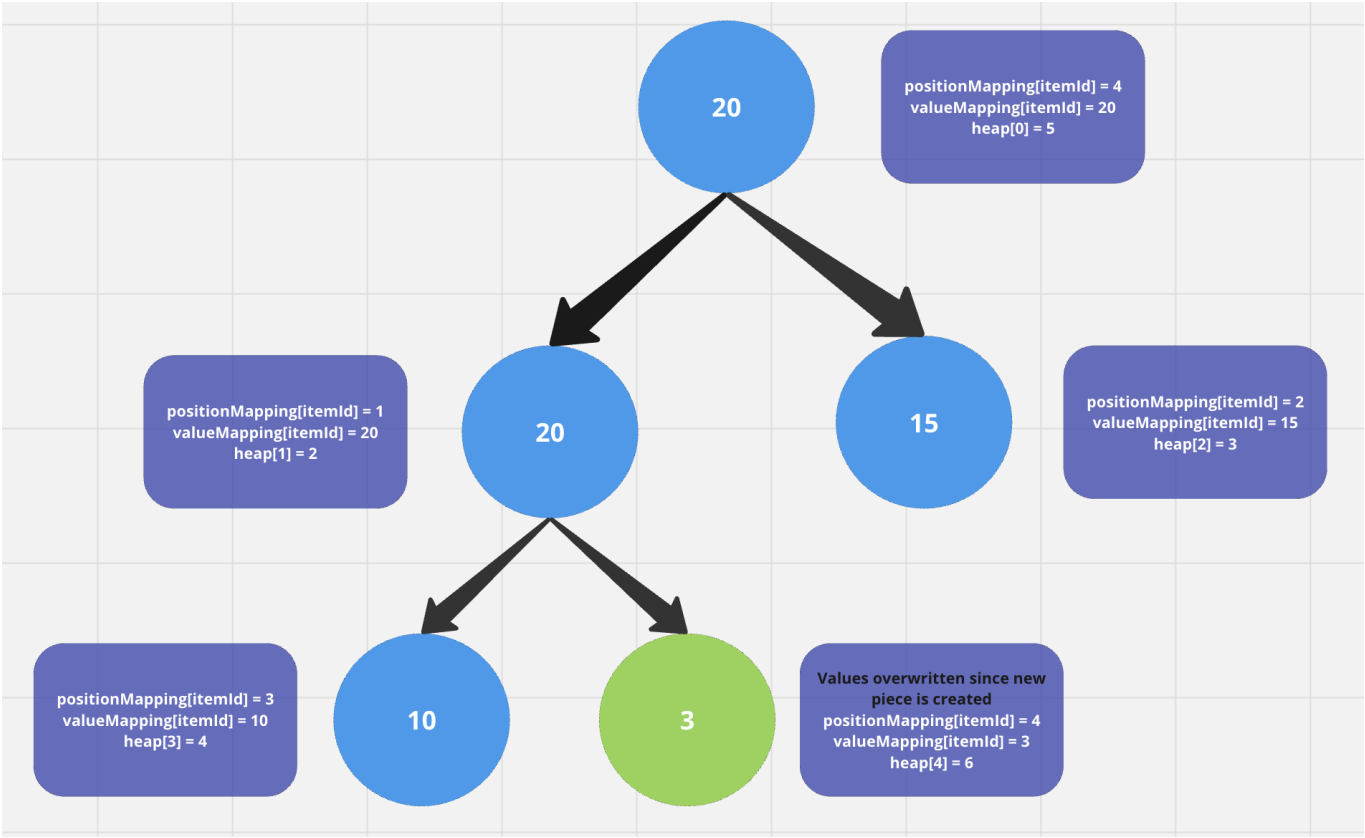
5. Now let's say an itemId 6 with a value of 3 is created using the insert() function. The following occurs:

- On Lines 131-133, heap[4] is updated to itemId 6, valueMapping[itemId] to 3 and positionMapping[itemId] to 4.

```
File: MaxHeap.sol
130:     function insert(uint256 itemId, uint256 value) public onlyAdmin {
131:         heap[size] = itemId;
132:         valueMapping[itemId] = value; // Update the value mapping
133:         positionMapping[itemId] = size; // Update the position
mapping
134:
135:         uint256 current = size;
136:         while (current != 0 && valueMapping[heap[current]] >
valueMapping[heap[parent(current)]]) {
137:             swap(current, parent(current));
138:             current = parent(current);
139:         }
140:         size++;
141:     }
```

6. If we look at the new state of the tree now, we can observe that both the itemId 5 at the root node and the latest itemId 6 share the same indexes in the tree i.e. positionMapping for both being 4.

Link to image



7. Now if we call the updateValue() function to update value for itemId 5 i.e. the root node, the following occurs:

- On Line 151, positionMapping[5] returns 4 as the position of the root itemId 5, which is clearly incorrect as demonstrated in point 6 above.
- On Line 152, oldValue stores the value of itemId 5 which is 20.
- Line 155 updates the valueMapping with the newValue.
- On Line 158, the check determines if the newValue is greater than the oldValue. Let's divide this into two cases, one for upward heapify and one for downward heapify.

```
File: MaxHeap.sol
150:     function updateValue(uint256 itemId, uint256 newValue) public
onlyAdmin {
151:         uint256 position = positionMapping[itemId];
152:         uint256 oldValue = valueMapping[itemId];
153:
154:         // Update the value in the valueMapping
155:         valueMapping[itemId] = newValue;
156:
157:         // Decide whether to perform upwards or downwards heapify
158:         if (newValue > oldValue) {
159:             // Upwards heapify
160:             while (position != 0 && valueMapping[heap[position]] >
valueMapping[heap[parent(position)]]) {
161:                 swap(position, parent(position));
162:                 position = parent(position);
163:             }
164:         } else if (newValue < oldValue) maxHeapify(position); //
```

```
Downwards heapify
165:     }
```

8. If we upward heapify by setting the newValue to be greater than the oldValue 20, the following occurs:

- On Line 160, the condition evaluates to false. Although expected since the root node is updated, the evaluation occurs incorrectly.
- The first condition checks if 4 != 0, which is true. This condition should actually have evaluated to false since the root node should have position/index = 0
- The second condition evaluates to false. This is because the valueMapping is accessing the value for itemId 6 and comparing it to its parent itemId 2.
- Due to this, although we do not enter the if block, the evaluation occurs incorrectly.

```
File: MaxHeap.sol
159:             // Upwards heapify
160:             while (position != 0 && valueMapping[heap[position]] >
valueMapping[heap[parent(position)]]) {
161:                 swap(position, parent(position));
162:                 position = parent(position);
163:             }
```

9. If we downward heapify by setting the newValue (let's say 13) to be lesser than the oldValue 20, the following occurs:

- On Line 164, we enter the else if block and make an internal call to maxHeapify() with position = 4 as argument instead of 0, which is incorrect.
- On Lines 101-102, left and right are set to indexes 9 and 10. This is incorrect and should have been 1 and 2 (children of the root node we are updating) instead of itemId 6's children.
- On Lines 103-105, itemId 6's values are retrieved i.e. 3, 0, 0 respectively.
- on Line 107, the condition evaluates to true since 4 >= 2 (i.e. size/2 = 5/2) and 4 <= 5 evaluate to true. Due to this we return early and the downward heapify does not occur.
- The impact here is that itemId 5 in the root now has a value of 13, which is **smaller than it's child node's values** i.e. 20 and 15 respectively. This not only breaks the max heap tree but also the binary tree's spec.
- Now if extractMax() is called, the root node itemId 5 with value 13 is popped instead of the expected itemId 2, which has a value of 20 and is the highest valued element.

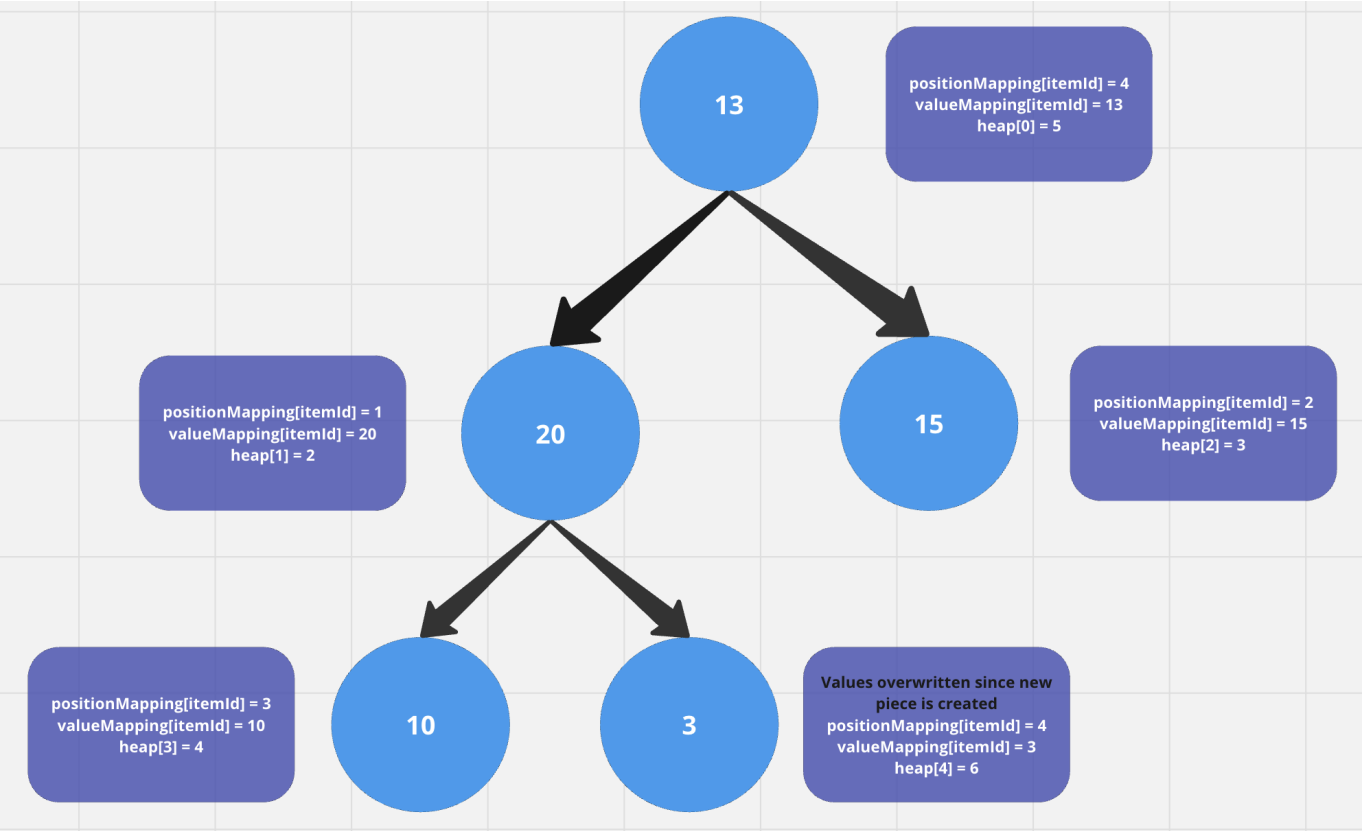
```
File: MaxHeap.sol
164:         } else if (newValue < oldValue) maxHeapify(position); //
Downwards heapify
```

maxHeapify() function:

```
File: MaxHeap.sol
099:     function maxHeapify(uint256 pos) internal {
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = 2 * pos + 2;
102:
103:         uint256 posValue = valueMapping[heap[pos]];
104:         uint256 leftValue = valueMapping[heap[left]];
105:         uint256 rightValue = valueMapping[heap[right]];
106:
107:         if (pos >= (size / 2) && pos <= size) return;
108:
109:         if (posValue < leftValue || posValue < rightValue) {
110:
111:             if (leftValue > rightValue) {
112:                 swap(pos, left);
113:                 maxHeapify(left);
114:             } else {
115:                 swap(pos, right);
116:                 maxHeapify(right);
117:             }
118:         }
119:     }
```

10. Here is how the max heap binary tree looks finally.

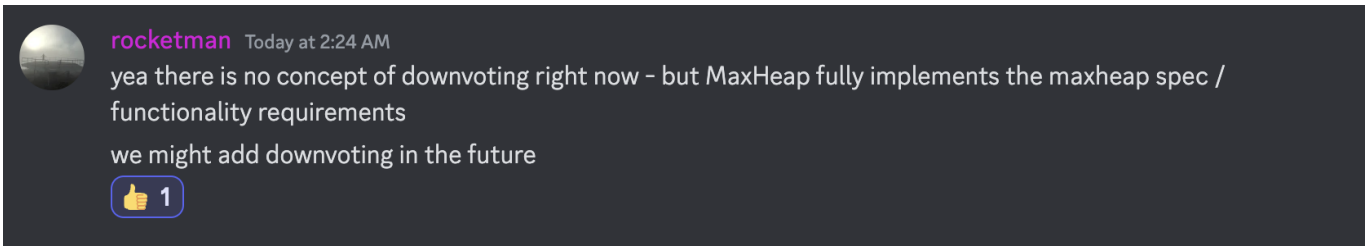
Link to image



Some points to note about this issue:

1. The downward heapifying issue has been demonstrated to display an additional impact to the already existing impact of incorrect indexing and max heap tree spec violation.
2. Although downward heapifying does not work in the codebase currently since downvoting does not exist, it can be introduced in the future based on sponsor's comments (see below). This would break the protocol functionality as demonstrated in this issue. This is because the data structure can change its admin to a future CultureIndex contract that supports downvoting.
3. Even if there is no concept of downvoting, MaxHeap is expected to fully implement the maxHeap spec and functionality requirements (see below), which it does not implement correctly.

Link to image



Tools Used

Manual Review

Recommended Mitigation Steps

The most straightforward solution to this would be to consider this type of case in the extractMax() function itself.

The following check ensures that if the last element is equal to the parent and it is greater than equal to the right index of the root node (i.e. index 2), we update the positionMapping correctly to 0. This check needs to be placed after this statement.

```
if (heap[size] == parent(size) && heap[size] >= heap[2]) {
   positionMapping[heap[0]] = 0;
}
```

[M-02] Last bidder can win auction by force sending ETH through selfdestruct() and not placing a new bid

Impact

After reservePrice has been updated mid-auction or towards the end of the auction, the last bidder's bid still exists and cannot be cancelled unless someone else places a higher bid above the new reservePrice. We assume bidders refrain from placing bids due to the high cost reservePrice now as well as end of the auction approaching,

In this case, the malicious last bidder can win the auction without placing a new bid. This is possible because this check here in the AuctionHouse.sol contract uses address(this).balance to determine whether

to burn an NFT or not. A malicious bidder can send ETH to the contract through selfdestruct() or a coinbase transaction. In both cases, the recipient will be set to the AuctionHouse.sol contract.

This way the bidder can use selfdestruct() (due to ease) to increase the balance of the AuctionHouse.sol contract and avoid entering this check here which refunds the ETH to the bidder and burns the NFT.

These are the following impacts:

1. Bidder wins the NFT without creating a bid
2. The auction.amount uses value less than the new updated reservePrice, which breaks the reservePrice invariant. 3, Since auction.amount is based on bidder's last bid, amount sent to creators, treasury and the other entities in ERC20TokenEmitter is way lesser than the expected amount. This means reduced rewards overall.
3. The extra ETH sent by the self-destruct operation remains locked in the AuctionHouse.sol contract and can never be retrieved by the team.
4. Although the bidder loses that locked amount of ETH, he not only won the auction while reducing rewards for the team but also can make more of the NFT that was bound to burn

Proof of Concept

Here is the whole process:

1. Let's assume the following

- reservePrice = 1 ETH
- auction.startTime = 100 (for example purposes)
- auction.endTime = 10000
- timeBuffer = 15 mins (based on tests)

2. After multiple bids by other bidders, bidder A places 3 ETH bid using createBid() function at time 9000.

- This sets the auction.amount to 3 ETH and bidder to A.
- The auction is extended by timeBuffer from current timestamp
- Refund to previous bidder is sent

```
File: AuctionHouse.sol
176:     function createBid(uint256 verbId, address bidder) external
payable override nonReentrant {
177:         IAuctionHouse.Auction memory _auction = auction;
178:
179:         //require bidder is valid address
180:         require(bidder != address(0), "Bidder cannot be zero
address");
181:         require(_auction.verbId == verbId, "Verb not up for
auction");
182:         //slither-disable-next-line timestamp
183:         require(block.timestamp < _auction.endTime, "Auction
expired");
184:         require(msg.value >= reservePrice, "Must send at least
reservePrice");
```

```
185:         require(
186:             msg.value >= _auction.amount + ((_auction.amount *
minBidIncrementPercentage) / 100),
187:             "Must send more than last bid by
minBidIncrementPercentage amount"
188:         );
189:
190:         address payable lastBidder = _auction.bidder;
191:
192:         auction.amount = msg.value;
193:         auction.bidder = payable(bidder);
194:
195:         // Extend the auction if the bid was received within
`timeBuffer` of the auction end time
196:         bool extended = _auction.endTime - block.timestamp <
timeBuffer;
197:         if (extended) auction.endTime = _auction.endTime =
block.timestamp + timeBuffer;
198:
199:         // Refund the last bidder, if applicable
200:         if (lastBidder != address(0))
_safeTransferETHWithFallback(lastBidder, _auction.amount);
201:
202:         emit AuctionBid(_auction.verbId, bidder, msg.sender,
msg.value, extended);
203:
204:         if (extended) emit AuctionExtended(_auction.verbId,
_auction.endTime);
205:     }
```

3. Following this, the reservePrice is updated to 4 ETH using the setReservePrice() function

```
File: AuctionHouse.sol
295:     function setReservePrice(uint256 _reservePrice) external override
onlyOwner {
296:         reservePrice = _reservePrice;
297:
298:         emit AuctionReservePriceUpdated(_reservePrice);
299:     }
```

4. This refrains bidders from further participating in the auction since the reservePrice now demands more than what they could have bid previously with the old reserve price and minBidIncrementPercentage.

5. During this period of timeBuffer (15 minutes), the malicious last bidder sends 1 ETH to this contract through selfdestruct() in his malicious contract that is destroyed. This will increase the address(this).balance of the AuctionHouse contract from 3 ETH to 4 ETH, which meets the reservePrice criteria. **(Note: Here auction.amount is still 3 ETH since a new bid is not being placed by the bidder)**.

6. Once the current timestamp exceeds the auction.endTime, the malicious bidder or anyone can call the settleCurrentAndCreateNewAuction() issue. This function internally calls _settleAuction(), where the following occurs:

- On Lines 347-350, we pass all checks since the auction is yet to be settled past the end time
- On Line 358, address(this).balance is checked to be less than reservePrice. If true, the bidder would be refunded his bid and the NFT would be burnt. But this why the malicious bidder sent ETH through selfdestruct() to manipulate this logic and not enter the if block. Due to this, we enter the else block.
- On Line 368, since auction.bidder is the malicious bidder, we enter the else block on Line 371 and the verb token is transferred to the malicious bidder.
- On Line 373, auction.amount is greater than 0, thus we enter the if block for payment distributions. But the auction.amount is 3 ETH instead of the expected 4 ETH since the amount sent through selfdestruct() is not accounted for in auction.amount. Due to this, the overall reward payment reduces by 1 ETH.

```
File: AuctionHouse.sol
344:     function _settleAuction() internal {
345:         IAuctionHouse.Auction memory _auction = auction;
346:
347:         require(_auction.startTime != 0, "Auction hasn't begun");
348:         require(!_auction.settled, "Auction has already been
settled");
349:         //slither-disable-next-line timestamp
350:         require(block.timestamp >= _auction.endTime, "Auction hasn't
completed");
351:
352:         auction.settled = true;
353:
354:         uint256 creatorTokensEmitted = 0;
355:         // Check if contract balance is greater than reserve price
356:
358:         if (address(this).balance < reservePrice) {
359:             // If contract balance is less than reserve price, refund
to the last bidder
360:             if (_auction.bidder != address(0)) {
361:                 _safeTransferETHWithFallback(_auction.bidder,
_auction.amount);
362:             }
363:
364:             // And then burn the Noun
365:             verbs.burn(_auction.verbId);
366:         } else {
367:             //If no one has bid, burn the Verb
368:             if (_auction.bidder == address(0))
369:                 verbs.burn(_auction.verbId);
370:                 //If someone has bid, transfer the Verb to the
winning bidder
371:             else verbs.transferFrom(address(this), _auction.bidder,
_auction.verbId);
372:
373:             if (_auction.amount > 0) {
```

```
374:                 // Ether going to owner of the auction
375:                 uint256 auctioneerPayment = (_auction.amount *
(10_000 - creatorRateBps)) / 10_000;
376:
377:                 //Total amount of ether going to creator
378:                 uint256 creatorsShare = _auction.amount -
auctioneerPayment;
```

Tools Used

Manual Review

Recommended Mitigation Steps

Instead of using address(this).balance, use auction.amount when comparing to reservePrice.

[M-03] Last element can be overwritten and removed from existence in max heap tree

Impact

During the extraction of the max element through the function extractMax(), the `positionMapping` for the last element in the heap tree is not updated when the last element is equal to its parent.

These are the following impacts:

1. MaxHeap does not function as intended and breaks its expected functionality due to element being incorrectly indexed in the heap.
2. When a new element is inserted to that index, the incorrectly indexed element accesses the new element's itemId during value updates
3. Downward heapifying will work incorrectly leading to the parent having a value smaller than its child. Thus, further breaking not only the maxHeap tree but also the binary tree spec.
4. Error in this heapifying will lead to the incorrectly indexed element being extracted.

Note: Point 3 in the impacts above breaks an invariant of the MaxHeap tree mentioned in the README.

```
The MaxHeap should always maintain the property of being a binary tree in
which the value in each internal node is greater than or equal to the
values in the children of that node.
```

Proof of Concept

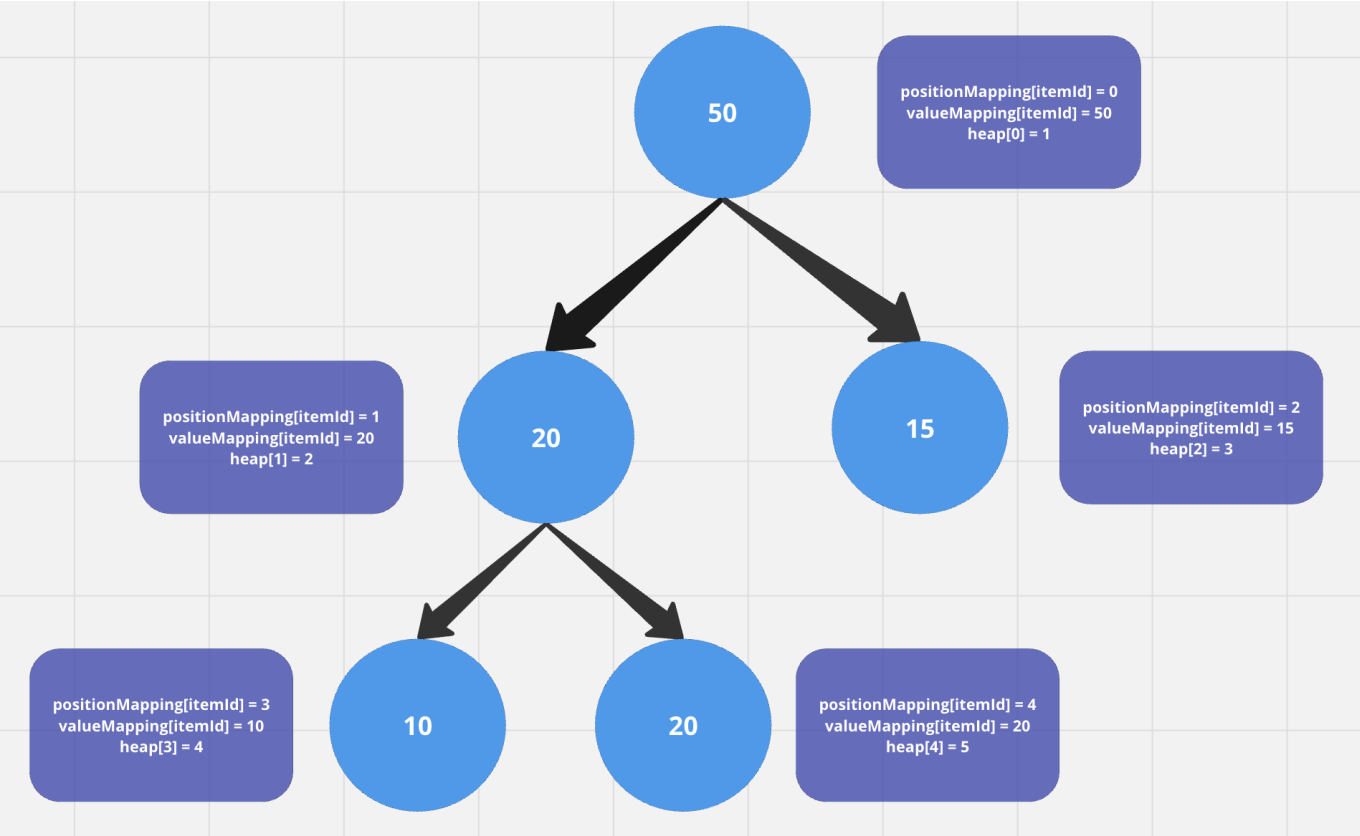
Here is the whole process:

1. Let's assume this is the current state of the maxHeap tree.

- The values have been made small for ease of demonstration of the issue
- The itemIds being used are 1,2,3,4,5 with values being 50,20,15,10,20 respectively.

- The current indexes of the items are 0,1,2,3,4 (since size starts from 0)
- The current size of the tree is 5.
- Over here, note that **itemId 2 is the parent of itemId 5 and both have the same values 20**.

Link to image



2. Now let's say the top element with itemId = 1 and value = 50 is extracted using the extractMax() function. The following occurs:

- On Line 174, we store the itemId at the root into the variable popped
- On Line 175, we replace the itemId at the root (effectively erasing it) with the last element in the heap tree i.e. itemId 5.
- On Line 175, size is decremented from 5 to 4 as expected.
- Following this on Line 178, we maxHeapify the current element at the root i.e. itemId 5 which was just set on Line 174.

```
File: MaxHeap.sol
171:     function extractMax() external onlyAdmin returns (uint256,
uint256) {
172:         require(size > 0, "Heap is empty");
173:
174:         uint256 popped = heap[0];
175:         heap[0] = heap[--size];
176:
177:
178:         maxHeapify(0);
179:
180:         return (popped, valueMapping[popped]);
181:     }
```

3. During the maxHeapify() internal function call, the following occurs:

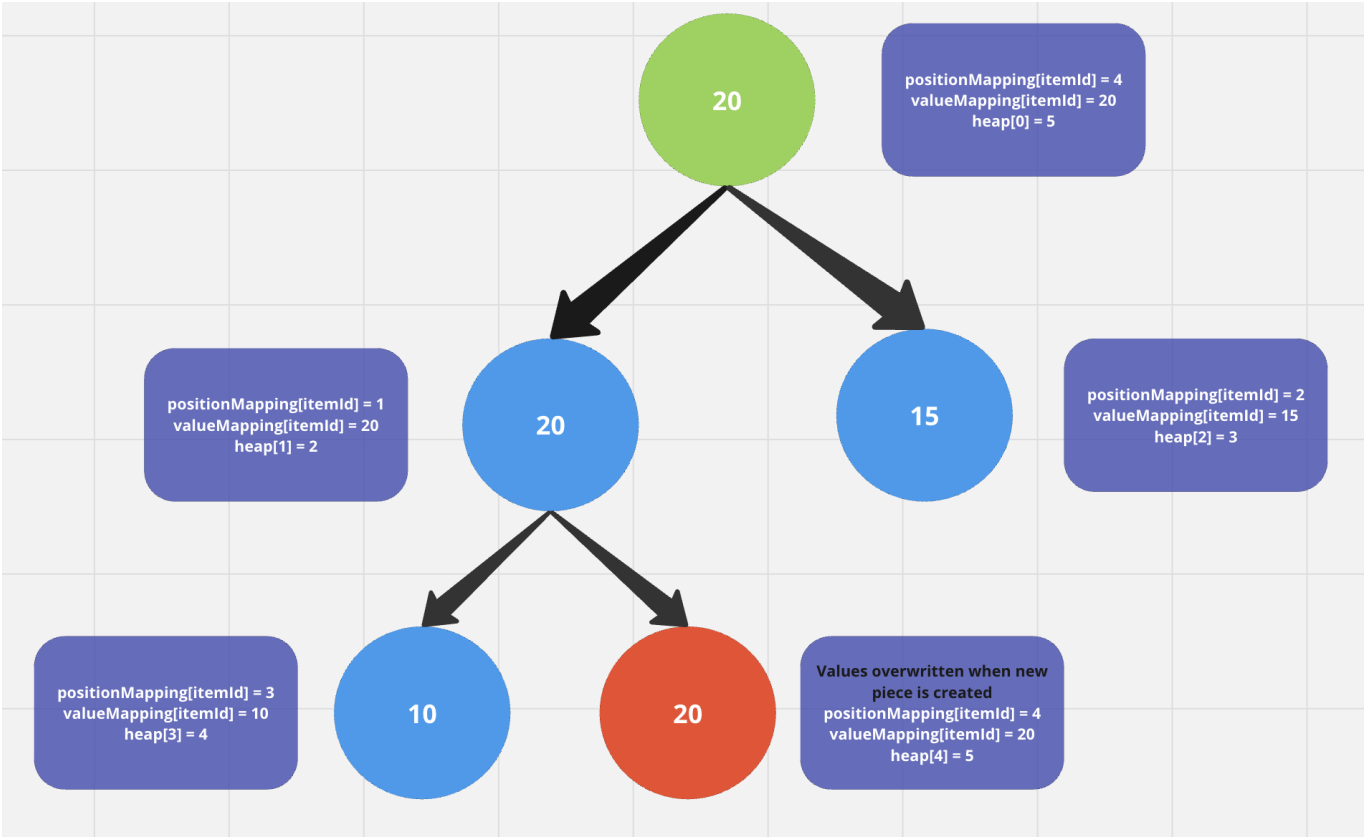
- On Lines 100-101, left and right are the indexes 1 and 2 (since pos = 0)
- On Lines 103-105, the values of the respective itemIds 5,2,3 are extracted as 20,20,15 respectively.
- On Line 107, we pass the check since 0 is not >= 2 (since size/2 = 4/2 = 2)
- On Line 109, we do not enter the if block since in the tree now, itemId 5 at the root has value 20 which is neither less than itemId 2 with value 20 nor itemId 3 with value 15. Thus, the function returns back to extractMax() and the call ends.

```
File: MaxHeap.sol
099:     function maxHeapify(uint256 pos) internal {
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = 2 * pos + 2;
102:
103:         uint256 posValue = valueMapping[heap[pos]];
104:         uint256 leftValue = valueMapping[heap[left]];
105:         uint256 rightValue = valueMapping[heap[right]];
106:
107:         if (pos >= (size / 2) && pos <= size) return;
108:
109:         if (posValue < leftValue || posValue < rightValue) {
110:
111:             if (leftValue > rightValue) {
112:                 swap(pos, left);
113:                 maxHeapify(left);
114:             } else {
115:                 swap(pos, right);
116:                 maxHeapify(right);
117:             }
118:         }
119:     }
```

4. On observing the new state of the tree, we can see the following:

- ItemId 5 with value 20 has been temporarily removed from index 4 in the tree until a new item is inserted.
- ItemId 5 with value 20 is now the root of the tree.
- If we look at the details of the root node, we can see that although the heap and valueMapping mappings have been updated correctly, **the positionMapping still points to index 4 for itemId 5**.
- This issue originates because during the maxHeapify() call, the itemId 5 is not less than either of it's child nodes as the condition in maxHeapify() demands on Line 109 above.
- This is the first impact on the maxHeap data structure since it does not index as expected.

Link to image



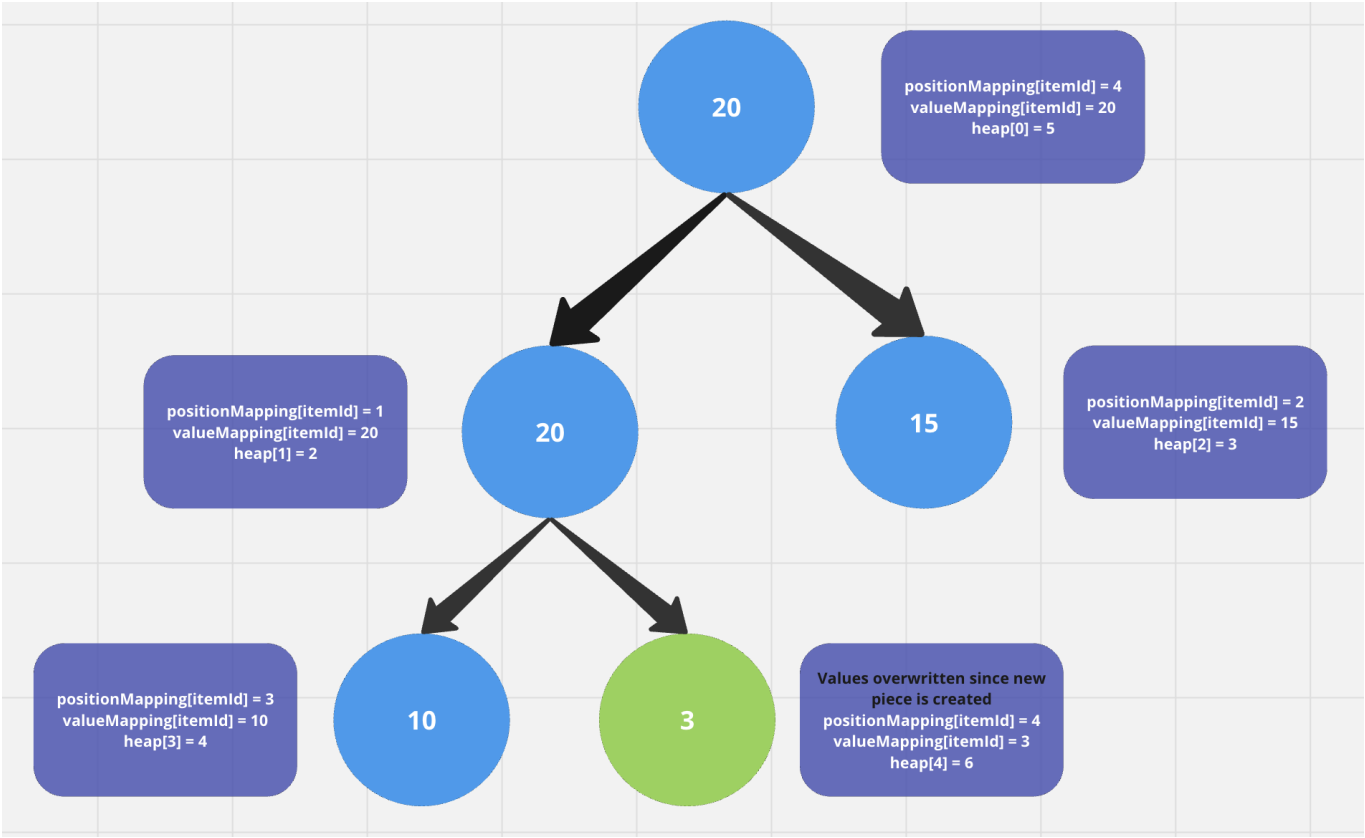
5. Now let's say an itemId 6 with a value of 3 is created using the insert() function. The following occurs:

- On Lines 131-133, heap[4] is updated to itemId 6, valueMapping[itemId] to 3 and positionMapping[itemId] to 4.

```
File: MaxHeap.sol
130:     function insert(uint256 itemId, uint256 value) public onlyAdmin {
131:         heap[size] = itemId;
132:         valueMapping[itemId] = value; // Update the value mapping
133:         positionMapping[itemId] = size; // Update the position
mapping
134:
135:         uint256 current = size;
136:         while (current != 0 && valueMapping[heap[current]] >
valueMapping[heap[parent(current)]]) {
137:             swap(current, parent(current));
138:             current = parent(current);
139:         }
140:         size++;
141:     }
```

6. If we look at the new state of the tree now, we can observe that both the itemId 5 at the root node and the latest itemId 6 share the same indexes in the tree i.e. positionMapping for both being 4.

Link to image



7. Now if we call the updateValue() function to update value for itemId 5 i.e. the root node, the following occurs:

- On Line 151, positionMapping[5] returns 4 as the position of the root itemId 5, which is clearly incorrect as demonstrated in point 6 above.
- On Line 152, oldValue stores the value of itemId 5 which is 20.
- Line 155 updates the valueMapping with the newValue.
- On Line 158, the check determines if the newValue is greater than the oldValue. Let's divide this into two cases, one for upward heapify and one for downward heapify.

```
File: MaxHeap.sol
150:     function updateValue(uint256 itemId, uint256 newValue) public
onlyAdmin {
151:         uint256 position = positionMapping[itemId];
152:         uint256 oldValue = valueMapping[itemId];
153:
154:         // Update the value in the valueMapping
155:         valueMapping[itemId] = newValue;
156:
157:         // Decide whether to perform upwards or downwards heapify
158:         if (newValue > oldValue) {
159:             // Upwards heapify
160:             while (position != 0 && valueMapping[heap[position]] >
valueMapping[heap[parent(position)]]) {
161:                 swap(position, parent(position));
162:                 position = parent(position);
163:             }
164:         } else if (newValue < oldValue) maxHeapify(position); //
```

```
Downwards heapify
165:     }
```

8. If we upward heapify by setting the newValue to be greater than the oldValue 20, the following occurs:

- On Line 160, the condition evaluates to false. Although expected since the root node is updated, the evaluation occurs incorrectly.
- The first condition checks if 4 != 0, which is true. This condition should actually have evaluated to false since the root node should have position/index = 0
- The second condition evaluates to false. This is because the valueMapping is accessing the value for itemId 6 and comparing it to its parent itemId 2.
- Due to this, although we do not enter the if block, the evaluation occurs incorrectly.

```
File: MaxHeap.sol
159:             // Upwards heapify
160:             while (position != 0 && valueMapping[heap[position]] >
valueMapping[heap[parent(position)]]) {
161:                 swap(position, parent(position));
162:                 position = parent(position);
163:             }
```

9. If we downward heapify by setting the newValue (let's say 13) to be lesser than the oldValue 20, the following occurs:

- On Line 164, we enter the else if block and make an internal call to maxHeapify() with position = 4 as argument instead of 0, which is incorrect.
- On Lines 101-102, left and right are set to indexes 9 and 10. This is incorrect and should have been 1 and 2 (children of the root node we are updating) instead of itemId 6's children.
- On Lines 103-105, itemId 6's values are retrieved i.e. 3, 0, 0 respectively.
- on Line 107, the condition evaluates to true since 4 >= 2 (i.e. size/2 = 5/2) and 4 <= 5 evaluate to true. Due to this we return early and the downward heapify does not occur.
- The impact here is that itemId 5 in the root now has a value of 13, which is **smaller than it's child node's values** i.e. 20 and 15 respectively. This not only breaks the max heap tree but also the binary tree's spec.
- Now if extractMax() is called, the root node itemId 5 with value 13 is popped instead of the expected itemId 2, which has a value of 20 and is the highest valued element.

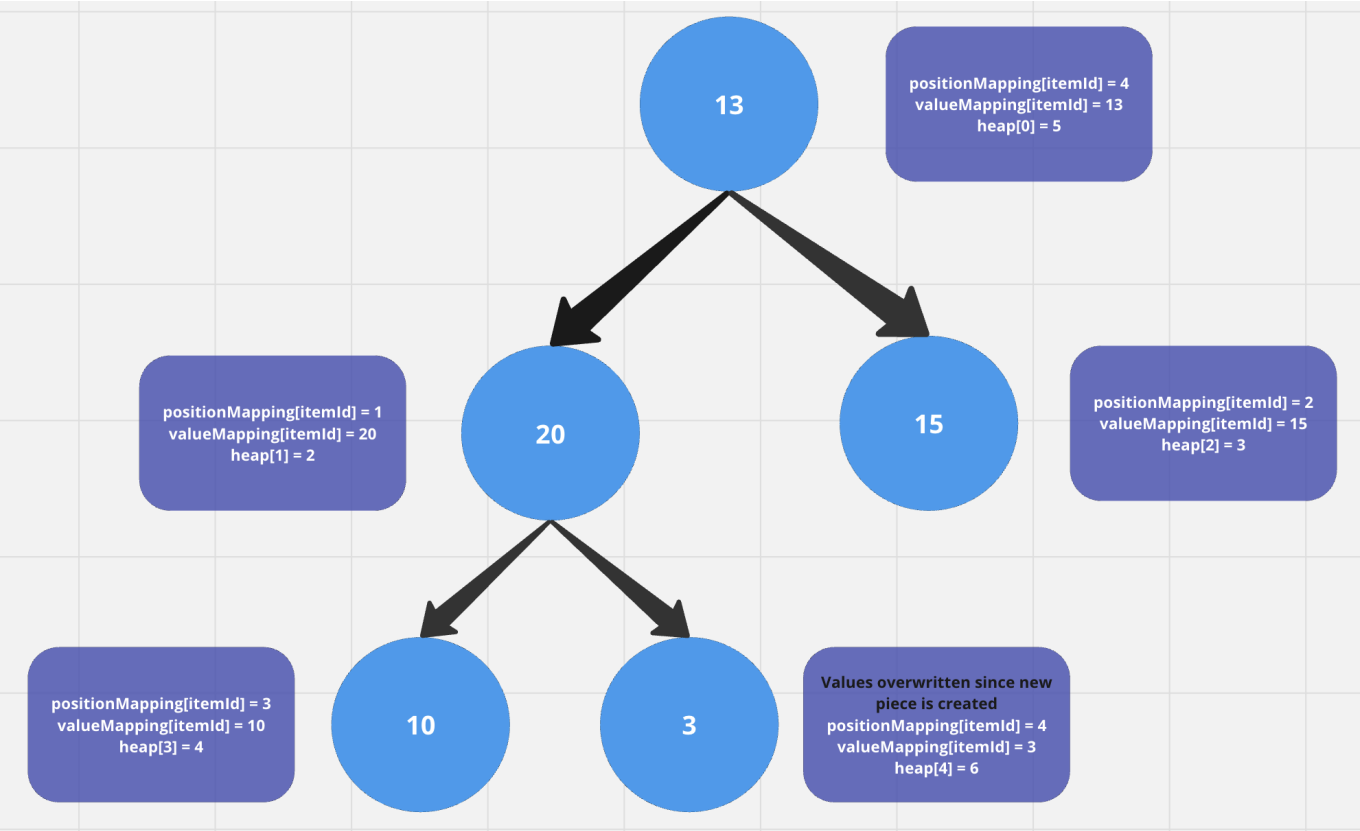
```
File: MaxHeap.sol
164:         } else if (newValue < oldValue) maxHeapify(position); //
Downwards heapify
```

maxHeapify() function:

```
File: MaxHeap.sol
099:     function maxHeapify(uint256 pos) internal {
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = 2 * pos + 2;
102:
103:         uint256 posValue = valueMapping[heap[pos]];
104:         uint256 leftValue = valueMapping[heap[left]];
105:         uint256 rightValue = valueMapping[heap[right]];
106:
107:         if (pos >= (size / 2) && pos <= size) return;
108:
109:         if (posValue < leftValue || posValue < rightValue) {
110:
111:             if (leftValue > rightValue) {
112:                 swap(pos, left);
113:                 maxHeapify(left);
114:             } else {
115:                 swap(pos, right);
116:                 maxHeapify(right);
117:             }
118:         }
119:     }
```

10. Here is how the max heap binary tree looks finally.

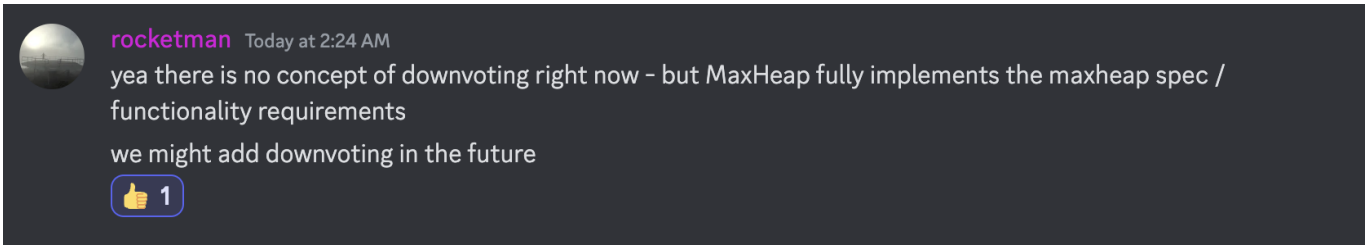
Link to image



Some points to note about this issue:

1. The downward heapifying issue has been demonstrated to display an additional impact to the already existing impact of incorrect indexing and max heap tree spec violation.
2. Although downward heapifying does not work in the codebase currently since downvoting does not exist, it can be introduced in the future based on sponsor's comments (see below). This would break the protocol functionality as demonstrated in this issue. This is because the data structure can change its admin to a future CultureIndex contract that supports downvoting but the CultureIndex cannot start using a new data structure since the max heap has existing data stored in it.
3. Even if there is no concept of downvoting, MaxHeap is expected to fully implement the maxHeap spec and functionality requirements (see below), which it does not implement correctly.

Link to image



Tools Used

Manual Review

Recommended Mitigation Steps

The most straightforward solution to this would be to consider this type of case in the extractMax() function itself.

The following check ensures that if the last element is equal to the parent and it is greater than equal to the right index of the root node (i.e. index 2), we update the positionMapping correctly to 0. This check needs to be placed after this statement.

```
if (heap[size] == parent(size) && heap[size] >= heap[2]) {
    positionMapping[heap[0]] = 0;
}
```

[N-01] Require check in getArtPieceById() function should use < instead of <=

Check should be < and not <= since mint happens following which an increment occurs for the next token which will be minted.

```
File: VerbsToken.sol
274:     function getArtPieceById(uint256 verbId) public view returns
(ICultureIndex.ArtPiece memory) {
275:         require(verbId <= _currentVerbId, "Invalid piece ID");
276:         return artPieces[verbId];
277:     }
```

[N-02] Redundant check in _mintTo() function can be removed

Check not required since when a piece is created, it undergoes this check.

```
File: VerbsToken.sol
287:         require(
288:             artPiece.creators.length <=
cultureIndex.MAX_NUM_CREATORS(),
289:             "Creator array must not be > MAX_NUM_CREATORS"
290:         );
```

[N-03] Incorrect comment for mapping heap

Correct struct to mapping in the notice tag on Line 64.

```
File: MaxHeap.sol
64:     /// @notice Struct to represent an item in the heap by it's ID
66:     mapping(uint256 => uint256) public heap;
```

[N-04] No need to initialize unsigned integer to 0

no need to initialize to 0 since it is the default value of unsigned integers.

```
File: MaxHeap.sol
67:     uint256 public size = 0;
```

[N-05] Remove redundant check from TokenEmitterRewards.sol

The check below will always return false i.e. we never revert since computeTotalReward() always returns 2.5% of the msgValue. Now since msgValue can never be less than 2.5% of itself, the check can be removed.

```
File: TokenEmitterRewards.sol
19:         if (msgValue < computeTotalReward(msgValue)) revert
INVALID_ETH_AMOUNT();
```

[L-01] PUSH0 opcode is not supported on BASE AND Optimism chains

All contracts in scope.txt use mostly v0.8.22

Make sure to either use version lower than 0.8.20 or set evm-version to Paris to deploy on the above mentioned chains.

[L-02] Prefer prioritizing the left hand side of the tree than right in binary tree

In the below code snippet, we can observe that if the leftValue > rightValue, the left side is preferred as expected. But for the case where leftValue = rightValue, currently the right hand side of the tree is prioritized when swapping. This can lead to some edge case scenarios where the property of ACBT (almost complete binary tree) can be broken. Thus use >= to prioritize the left side of the tree.

```
File: MaxHeap.sol
109:             if (leftValue > rightValue) {
110:                 swap(pos, left);
111:                 maxHeapify(left);
112:             } else {
113:                 swap(pos, right);
114:                 maxHeapify(right);
115:             }
```

[L-03] createPiece() can be spammed on cheap fee chains like Base and Optimism

Currently, there is not limit on how many pieces can be inserted into the max heap tree. This can lead to a DOS in the max heap tree down the line (due to unnecessary mountain of heapify operations required to be performed by legitimate new elements inserted) and frontend spamming if too many pieces are brought into existence. Additionally, since this codebase will be deployed on Base and Optimism, which have cheaper fees, the resources to perform such an attack increases.

To resolve this issue, consider adding a rate limiter per address, which would limit users from creating more pieces in a certain amount of time. For example, not more than 20-25 pieces per day.

[L-04] Return value of extractMax() not checked in function dropTopVotedPiece() in CultureIndex

The extractMax() function returns two uint256s i.e. the itemId at the top and its value, but neither of them are checked in the dropTopVotedPiece() function.

```
File: MaxHeap.sol
160:     function extractMax() external onlyAdmin returns (uint256,
uint256) {
161:         require(size > 0, "Heap is empty");
162:
163:         uint256 popped = heap[0];
164:         heap[0] = heap[--size];
165:         maxHeapify(0);
166:
167:         return (popped, valueMapping[popped]);
168:     }
```

[L-05] 1.75% of ETH provided to buyToken() can be refunded by users to themselves when buying the ERC20 tokens

Buyers of the ERC20 tokens can pass in their addresses as protocolRewardsRecipients. This would refund them their 1.75% of ETH through the ProtocolRewards contract.

```
File: ERC20TokenEmitter.sol
152:     function buyToken(
153:         address[] calldata addresses,
154:         uint[] calldata basisPointSplits,
155:         ProtocolRewardAddresses calldata protocolRewardsRecipients
156:     ) public payable nonReentrant whenNotPaused returns (uint256
tokensSoldWad) {
```

[L-06] Missing field and check for audio metadata type length to be greater than 0

Missing field in metadata for audio as "string audio" and thus missing check for it in the function below.

```
File: CultureIndex.sol
159:     function validateMediaType(ArtPieceMetadata calldata metadata)
internal pure {
160:         require(uint8(metadata.mediaType) > 0 &&
uint8(metadata.mediaType) <= 5, "Invalid media type");
161:
162:         if (metadata.mediaType == MediaType.IMAGE)
163:             require(bytes(metadata.image).length > 0, "Image URL must
be provided");
164:         else if (metadata.mediaType == MediaType.ANIMATION)
165:             require(bytes(metadata.animationUrl).length > 0,
"Animation URL must be provided");
166:         else if (metadata.mediaType == MediaType.TEXT)
167:             require(bytes(metadata.text).length > 0, "Text must be
provided");
168:
169:     }
```

[L-07] ERC20Votes liquidity fragmentation and art theft can occur if contracts are deployed across three different chains

The issue is that currently all ERC20Votes tokens are non transferrable, thus limiting their usage as wrapped assets for bridging across chains. Each of the contracts will have their independent states, which allows for art pieces to be copied and sold across other chains.

For art theft - consider adding artist signatures as a part of the ArtPieceMetadata.

[L-08] Use >= instead of > when using minVoteWeight as criteria to start voting

Minimum values usually represent the value required to cast a vote. But currently only > is considered instead of >=. This breaks a soft invariant of the protocol indirectly. Additionally users having the minimum amount of voting weight will not be allowed to vote, which is a loss for the creators of the art piece and the overall community supporting that art piece.

```
File: CultureIndex.sol
316: require(weight > minVoteWeight, "Weight must be greater than
minVoteWeight");
```

[L-09] Code does not follow spec and breaks soft invariant in pause/unpause functionalities

The function pause() can be called even when the function is already paused. The same goes for unpause(). Thus, this does not follow the expected code spec.

```
File: AuctionHouse.sol
203:     /**
204:      * @notice Pause the Verbs auction house.
205:      * @dev This function can only be called by the owner when the
206:      * contract is unpaused. While no new auctions can be started
when paused,
207:      * anyone can settle an ongoing auction.
208:      */
209:
210:     function pause() external override onlyOwner {
211:         _pause();
212:     }
```

[R-01] Expose burn() function with access control in NontransferableERC20Votes.sol for any future supply mechanism purposes

Currently in the NontransferableERC20Votes contract, only the mint() function has been externally exposed, which is called from the ERC20TokenEmitter. Although not necessary, it would be good to expose the burn() function with the onlyOwner access control as well, which could be used as some mintpass mechanism or supply mechanism in the future.

[R-02] Consider removing the `_minCreatorRateBps > minCreatorRateBps` check

The below check should be removed in function setMinCreatorRateBps() since it limits the new minimum to always be greater. This is not a logical check since the the function itself is used to set the minCreatorRateBps.

```
File: AuctionHouse.sol
241:         require(
242:             _minCreatorRateBps > minCreatorRateBps,
```

```
243:             "Min creator rate must be greater than previous
minCreatorRateBps"
244:         );
```

Gas Optimizations

Optimizations have been focused on specific contracts/functions as requested by the sponsor in the README here under Gas Reports.

[G-01] `MAX_NUM_CREATORS` check is not required when minting in VerbsToken.sol contract

The check below in the _mintTo() function can be removed since it is already checked when creating a piece in CultureIndex.sol (see here).

```
File: VerbsToken.sol
288:         require(
289:             artPiece.creators.length <=
cultureIndex.MAX_NUM_CREATORS(),
290:             "Creator array must not be > MAX_NUM_CREATORS"
291:         );
```

[G-02] Mappings not used externally/internally can be marked private

The below mappings from the MaxHeap.sol contract can be marked private since they're not used by any other contracts.

```
File: MaxHeap.sol
66:     /// @notice Struct to represent an item in the heap by it's ID
67:     mapping(uint256 => uint256) public heap;
68:
69:     uint256 public size = 0;
70:
71:     /// @notice Mapping to keep track of the value of an item in the
heap
72:     mapping(uint256 => uint256) public valueMapping;
73:
74:     /// @notice Mapping to keep track of the position of an item in
the heap
75:     mapping(uint256 => uint256) public positionMapping;
```

[G-03] No need to initialize variable `size` to 0

0 is the default value for an unsigned integer, thus setting it to 0 again is not required.

```
File: MaxHeap.sol
69:     uint256 public size = 0;
```

[G-04] Use `left + 1` to calculate value of `right` in `maxHeapify()` to save gas

The variable `right` can be re-written as 2 * pos + 1 + 1. Since we already know `left` is 2 * pos + 1, we can use left + 1 to save gas. This would mainly remove the MUL opcode (5 gas) and CALLDATALOAD (3 gas) and add just an MLOAD opcode (3 gas) instead. Thus saving a net of 5 gas per call.

Instead of this:

```
File: MaxHeap.sol
099:     function maxHeapify(uint256 pos) internal {
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = 2 * pos + 2; //@audit Gas - Just use left + 1
```

Use this:

```
File: MaxHeap.sol
099:     function maxHeapify(uint256 pos) internal {
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = left + 1;
```

[G-05] Place size check at the start of maxHeapify() to save gas on returning case

The check on Line 107 can be moved to the start of the maxHeapify() function. This will save gas when the condition becomes true because the computations from Line 100 to 105 will not be executed for the returning case since we will return early.

Instead of this:

```
File: MaxHeap.sol
099:     function maxHeapify(uint256 pos) internal {
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = 2 * pos + 2;
102:
103:         uint256 posValue = valueMapping[heap[pos]];
104:         uint256 leftValue = valueMapping[heap[left]];
105:         uint256 rightValue = valueMapping[heap[right]];
106:
107:         if (pos >= (size / 2) && pos <= size) return;
109:
111:         if (posValue < leftValue || posValue < rightValue) {
```

```
113:
114:             if (leftValue > rightValue) {
115:                 swap(pos, left);
116:                 maxHeapify(left);
117:             } else {
118:                 swap(pos, right);
119:                 maxHeapify(right);
120:             }
121:         }
122:     }
```

Use this (see line 99):

```
File: MaxHeap.sol
098:     function maxHeapify(uint256 pos) internal {
099:         if (pos >= (size / 2) && pos <= size) return;
100:         uint256 left = 2 * pos + 1;
101:         uint256 right = 2 * pos + 2;
102:
103:         uint256 posValue = valueMapping[heap[pos]];
104:         uint256 leftValue = valueMapping[heap[left]];
105:         uint256 rightValue = valueMapping[heap[right]];
106:
109:
111:         if (posValue < leftValue || posValue < rightValue) {
113:
114:             if (leftValue > rightValue) {
115:                 swap(pos, left);
116:                 maxHeapify(left);
117:             } else {
118:                 swap(pos, right);
119:                 maxHeapify(right);
120:             }
121:         }
122:     }
```

[G-06] Cache `parent(current)` in insert() function to save gas

Link to instance Link to another instance

Caching the parent() function call will save an unnecessary JUMPDEST and internal operations in the function itself.

Instead of this:

```
File: MaxHeap.sol
138:             swap(current, parent(current));
139:             current = parent(current);
```

Use this:

```
File: MaxHeap.sol
137:             uint256 parentOfCurrent = parentCurrent(current);
138:             swap(current, parentOfCurrent);
139:             current = parentOfCurrent;
```

[G-07] else-if block in function updateValue() can be removed since newValue can never be less than oldValue

The else-if block below is present in the updateValue() function. This can be removed because votes made from function _vote() cannot be cancelled or decreased, thus this case is never true. This will save both function execution cost and deployment cost.

```
File: MaxHeap.sol
164:         } else if (newValue < oldValue) maxHeapify(position);
```

[G-08] `size > 0` check not required in function getMax()

The check `size > 0` is already implemented in the function topVotedPieceId() here (which is accessed from dropTopVotedPiece()) in CultureIndex.sol, thus implementing it in getMax() function in MaxHeap.sol is not required.

```
File: MaxHeap.sol
188:         require(size > 0, "Heap is empty");
```

[G-09] Cache `msgValueRemaining - toPayTreasury` in buyToken() to save gas

The same subtraction operation with the same variables is carried out in three different steps. Consider caching this value to save gas.

Instead of this:

```
File: ERC20TokenEmitter.sol
178:         uint256 creatorDirectPayment = ((msgValueRemaining -
toPayTreasury) * entropyRateBps) / 10_000;
179:
180:         //Tokens to emit to creators
181:         int totalTokensForCreators = ((msgValueRemaining -
toPayTreasury) - creatorDirectPayment) > 0
182:             ? getTokenQuoteForEther((msgValueRemaining -
toPayTreasury) - creatorDirectPayment)
183:             : int(0);
```

Use this (see Line 177):

```
File: ERC20TokenEmitter.sol
177:         uint256 cachedValue = msgValueRemaining - toPayTreasury;
178:         uint256 creatorDirectPayment = ((cachedValue) *
entropyRateBps) / 10_000;
179:
180:         //Tokens to emit to creators
181:         int totalTokensForCreators = ((cachedValue) -
creatorDirectPayment) > 0
182:             ? getTokenQuoteForEther((cachedValue) -
creatorDirectPayment)
183:             : int(0);
```

[G-10] `creatorsAddress != address(0)` check not required in buyToken()

The check below is from the buyToken() function (see here). The second condition `creatorsAddress != address(0)` can be removed since `creatorsAddress` can never be the zero address. This is because function setCreatorsAddress() implements this check already.

```
File: ERC20TokenEmitter.sol
205:         if (totalTokensForCreators > 0 && creatorsAddress !=
address(0)) {
```

[G-11] Cache return value of _calculateTokenWeight() function to prevent SLOAD

In the function createPiece() here, the totalVotesSupply returned can be cached into a memory variable and then assigned to Line 229 and 237. This will prevent the unnecessary SLOAD of `newPiece.totalVoteSupply` on Line 237.

```
File: CultureIndex.sol
229:         newPiece.totalVotesSupply = _calculateVoteWeight(
230:             erc20VotingToken.totalSupply(),
231:             erc721VotingToken.totalSupply()
232:         );
233:         newPiece.totalERC20Supply = erc20VotingToken.totalSupply();
234:         newPiece.metadata = metadata;
235:         newPiece.sponsor = msg.sender;
236:         newPiece.creationBlock = block.number;
237:         newPiece.quorumVotes = (quorumVotesBPS *
newPiece.totalVotesSupply) / 10_000;
```

[G-12] Cache `erc20VotingToken.totalSupply()` to save gas

The value of the call `erc20VotingToken.totalSupply()` is used on Line 230 and Line 233. This can save an unnecessary totalSupply() call on the erc20VotingToken if cached.

```
File: CultureIndex.sol
229:         newPiece.totalVotesSupply = _calculateVoteWeight(
230:             erc20VotingToken.totalSupply(),
231:             erc721VotingToken.totalSupply()
232:         );
233:         newPiece.totalERC20Supply = erc20VotingToken.totalSupply();
```

[G-13] Unnecessary for loop can be removed by shifting its statements into an existing for loop

The for loop on Line 247 is not required since the for loop on Line 239 already loops through the same range [0, creatorArrayLength). Thus the for loop on Line 247 can be removed and the event emissions of PieceCreatorAdded() can be moved into the for loop on Line 239. This would save gas on both deployment and during function execution.

```
File: CultureIndex.sol
239:         for (uint i; i < creatorArrayLength; i++) {
240:             newPiece.creators.push(creatorArray[i]);
241:         }
242:
243:         emit PieceCreated(pieceId, msg.sender, metadata,
newPiece.quorumVotes, newPiece.totalVotesSupply);
244:
245:         // Emit an event for each creator
246:
247:         for (uint i; i < creatorArrayLength; i++) {
248:             emit PieceCreatorAdded(pieceId, creatorArray[i].creator,
msg.sender, creatorArray[i].bps);
249:         }
```

[G-14] Return memory variable `pieceId` instead of storage variable `newPiece.pieceId` to save gas

Return memory variable `pieceId` instead of accessing value from storage and returning. This would replace the SLOAD (100 gas) with an MLOAD (3 gas) operation.

```
File: CultureIndex.sol
250:         return newPiece.pieceId;
```

[G-15] Calculate `creatorsShare` before `auctioneerPayment` in buyToken() to prevent unnecessary SUB operation

On Line 388, when calculating the `auctioneerPayment`, we find out the bps for the auctioneer by subtracting 10000 from creatorRateBps. This SUB operation will not be required if we calculate the `creatorsShare` on Line 391 before the `auctioneerPayment`. In this case the `creatorsShare` can just be calculated using `_auction.amount * creatorRateBps / 10000`. Following which the `auctioneerPayment` can just be calculated using `_auction.amount - creatorsShare` (as done by creatorsShare previously on Line 391). Through this we can see a redundant SUB operation can be removed which helps save gas.

```
File: AuctionHouse.sol
388:                 uint256 auctioneerPayment = (_auction.amount *
(10_000 - creatorRateBps)) / 10_000;
389:
390:                 //Total amount of ether going to creator
391:                 uint256 creatorsShare = _auction.amount -
auctioneerPayment;
```

[G-16] Remove `msgValue < computeTotalReward(msgValue` check from TokenEmitterRewards.sol contract

On Line 18, when passing msgValue as parameter to computeTotalReward(), we will always pass this check because computeTotalReward always returns 2.5% of the msgValue. Thus since msgValue can never be less than 2.5% of itself, the condition never evaluates to true and we never revert.

```
File: TokenEmitterRewards.sol
12:     function _handleRewardsAndGetValueToSend(
13:         uint256 msgValue,
14:         address builderReferral,
15:         address purchaseReferral,
16:         address deployer
17:     ) internal returns (uint256) {
18:         if (msgValue < computeTotalReward(msgValue)) revert
INVALID_ETH_AMOUNT();
19:
20:         return msgValue - _depositPurchaseRewards(msgValue,
builderReferral, purchaseReferral, deployer);
21:     }
```

[G-17] Optimize `computeTotalReward()` and `computePurchaseRewards` into one function to save gas

Both the functions below do the same computation, which is not required if the functions are combined into one. This will also prevent the rounding issue mentioned here.

Instead of this:

```
File: RewardSplits.sol
41:     function computeTotalReward(uint256 paymentAmountWei) public pure
returns (uint256) {
42:         if (paymentAmountWei <= minPurchaseAmount || paymentAmountWei
>= maxPurchaseAmount) revert INVALID_ETH_AMOUNT();
43:
45:         return
46:             (paymentAmountWei * BUILDER_REWARD_BPS) /
47:             10_000 +
48:             (paymentAmountWei * PURCHASE_REFERRAL_BPS) /
49:             10_000 +
50:             (paymentAmountWei * DEPLOYER_REWARD_BPS) /
51:             10_000 +
52:             (paymentAmountWei * REVOLUTION_REWARD_BPS) /
53:             10_000;
54:     }
55:
56:     function computePurchaseRewards(uint256 paymentAmountWei) public
pure returns (RewardsSettings memory, uint256) {
57:         return (
58:             RewardsSettings({
59:                 builderReferralReward: (paymentAmountWei *
BUILDER_REWARD_BPS) / 10_000,
60:                 purchaseReferralReward: (paymentAmountWei *
PURCHASE_REFERRAL_BPS) / 10_000,
61:                 deployerReward: (paymentAmountWei *
DEPLOYER_REWARD_BPS) / 10_000,
62:                 revolutionReward: (paymentAmountWei *
REVOLUTION_REWARD_BPS) / 10_000
63:             }),
64:             computeTotalReward(paymentAmountWei)
65:         );
66:     }
```

Use this:

```
File: RewardSplits.sol
41:     function computeTotalPurchaseRewards(uint256 paymentAmountWei)
public pure returns (RewardsSettings memory, uint256) {
42:         if (paymentAmountWei <= minPurchaseAmount || paymentAmountWei
>= maxPurchaseAmount) revert INVALID_ETH_AMOUNT();
43:
44:         uint256 br = (paymentAmountWei * BUILDER_REWARD_BPS) / 10_000;
45:         uint256 pr = (paymentAmountWei * PURCHASE_REFERRAL_BPS) /
10_000;
46:         uint256 dr = (paymentAmountWei * DEPLOYER_REWARD_BPS) /
10_000;
47:         uint256 rr = (paymentAmountWei * REVOLUTION_REWARD_BPS) /
10_000;
48:
```

```
49:         return (
50:             RewardsSettings({
51:                 br,
52:                 pr,
53:                 dr,
54:                 rr
55:             }),
56:             br + pr + dr + rr
57:         );
58:     }
```

[G-18] Calculation in computeTotalReward() can be simplified to save gas

Currently the calculation is very repetitive as seen below and some similar operations occur internally. If we observe closely, each reward separated by + have paymentAmountWei and 10000 common in them.

Let us assume x = paymentAmountWei, y = 10000 and a,b,c,d = each of the reward bps types respectively.

Current equation = (x * a)/y + (x * b)/y + (x * c)/y + (x * d)/y

Let's take x / y common,

Modified Equation = x/y * a + x/y * b + x/y * c + x/y * d

Further let's take x/y common from all terms,

Modified Equation = x/y * (a + b + c + d)

Now since we need to get rid of the rounding, we just multiply first instead of divide.

Final Equation = (x * (a + b + c + d)) / y

This final equation will save us alot of gas without compromising on any rounding issues.

Instead of this:

```
File: RewardSplits.sol
41:     function computeTotalReward(uint256 paymentAmountWei) public pure
returns (uint256) {
42:         if (paymentAmountWei <= minPurchaseAmount || paymentAmountWei
>= maxPurchaseAmount) revert INVALID_ETH_AMOUNT();
43:
44:
45:         return
46:             (paymentAmountWei * BUILDER_REWARD_BPS) /
47:             10_000 +
48:             (paymentAmountWei * PURCHASE_REFERRAL_BPS) /
49:             10_000 +
50:             (paymentAmountWei * DEPLOYER_REWARD_BPS) /
51:             10_000 +
52:             (paymentAmountWei * REVOLUTION_REWARD_BPS) /
```

```
53:             10_000;
54:     }
```

Use this:

```
File: RewardSplits.sol
41:     function computeTotalReward(uint256 paymentAmountWei) public pure
returns (uint256) {
42:         if (paymentAmountWei <= minPurchaseAmount || paymentAmountWei
>= maxPurchaseAmount) revert INVALID_ETH_AMOUNT();
43:
44:
45:         return
46:             (paymentAmountWei * (BUILDER_REWARD_BPS +
PURCHASE_REFERRAL_BPS + DEPLOYER_REWARD_BPS + REVOLUTION_REWARD_BPS)) /
10_000;
47:     }
```

[G-19] Negating twice in require check is not required in _vote() function

The require check below can just use == instead of negating twice to ensure voter has already voted or not.

Instead of this:

```
File: CultureIndex.sol
315:         require(!(votes[pieceId][voter].voterAddress != address(0)),
"Already voted");
```

Use this:

```
File: CultureIndex.sol
315:         require(votes[pieceId][voter].voterAddress == address(0),
"Already voted");
```