Community Seminar: Clober

Audit Details

- Timeline: Jan 2 - Jan 13 +2.5 week fix period
- Team: Cmichel, HickupHH3, Throttle, Taek, Grmpyninja
- Launch: 13th Feb, initially on Polygon, then on other networks

Notable Features

- On-chain orderbook that aims to reduce linear overhead of taker orders
- Each price index (tick) can store up to 2^{15} orders
- Track the aggregated liquidity at that price level instead of iterating through each maker order
- Custom data structures: Segmented Segment Tree to track liquidity depth & Octopus Heap to track initialised price indexes

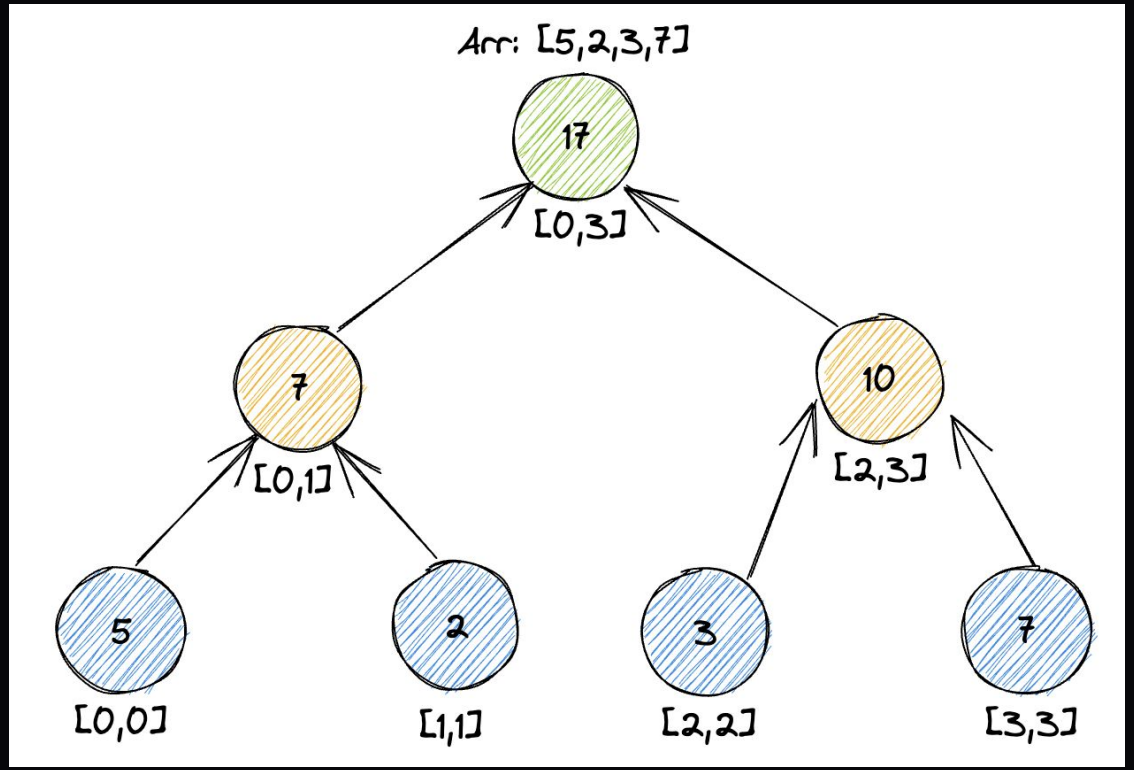
Notable Features

- FIFO: earlier orders get filled first
- Order settlement only transfers tokens to the taker, makers' filled orders have to claim separately
- Maker orders are represented by NFTs: transferring NFTs will also transfer claiming rights

Findings & Learning Points

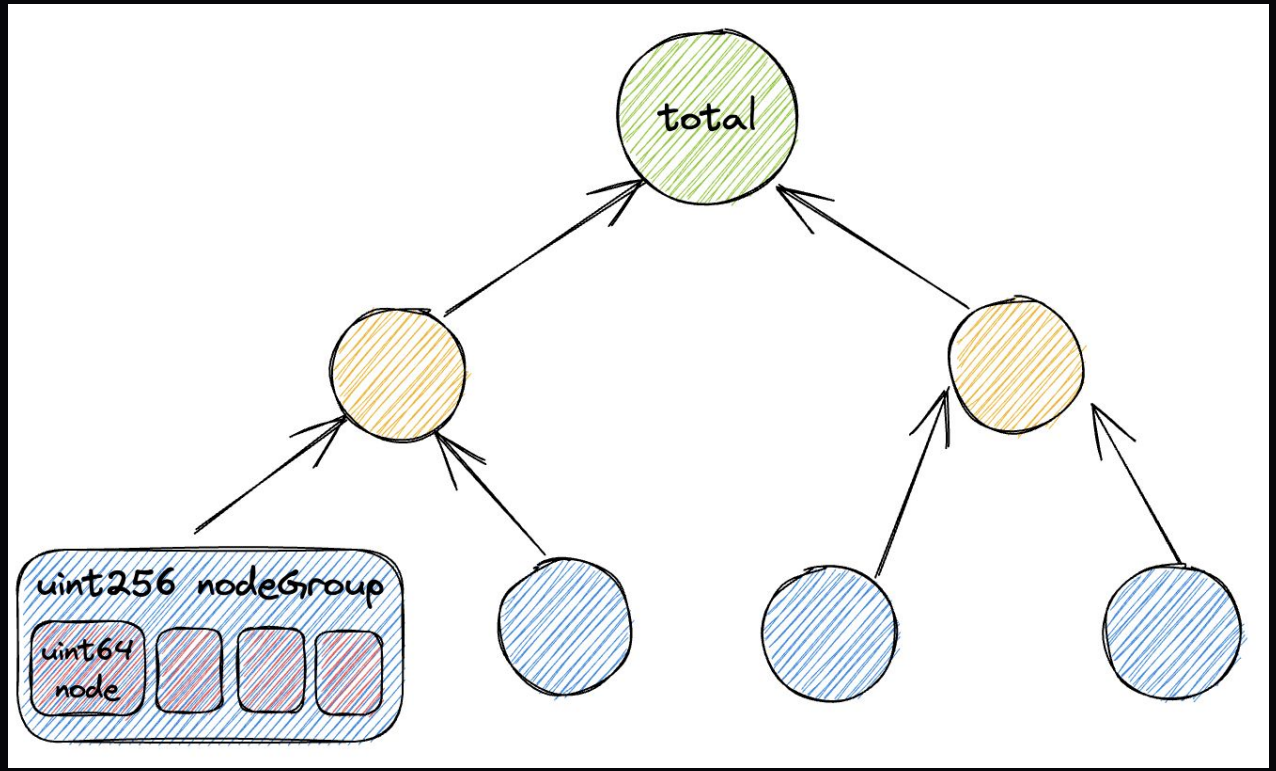
Addition overflow in Segmented Segment Tree

Crit



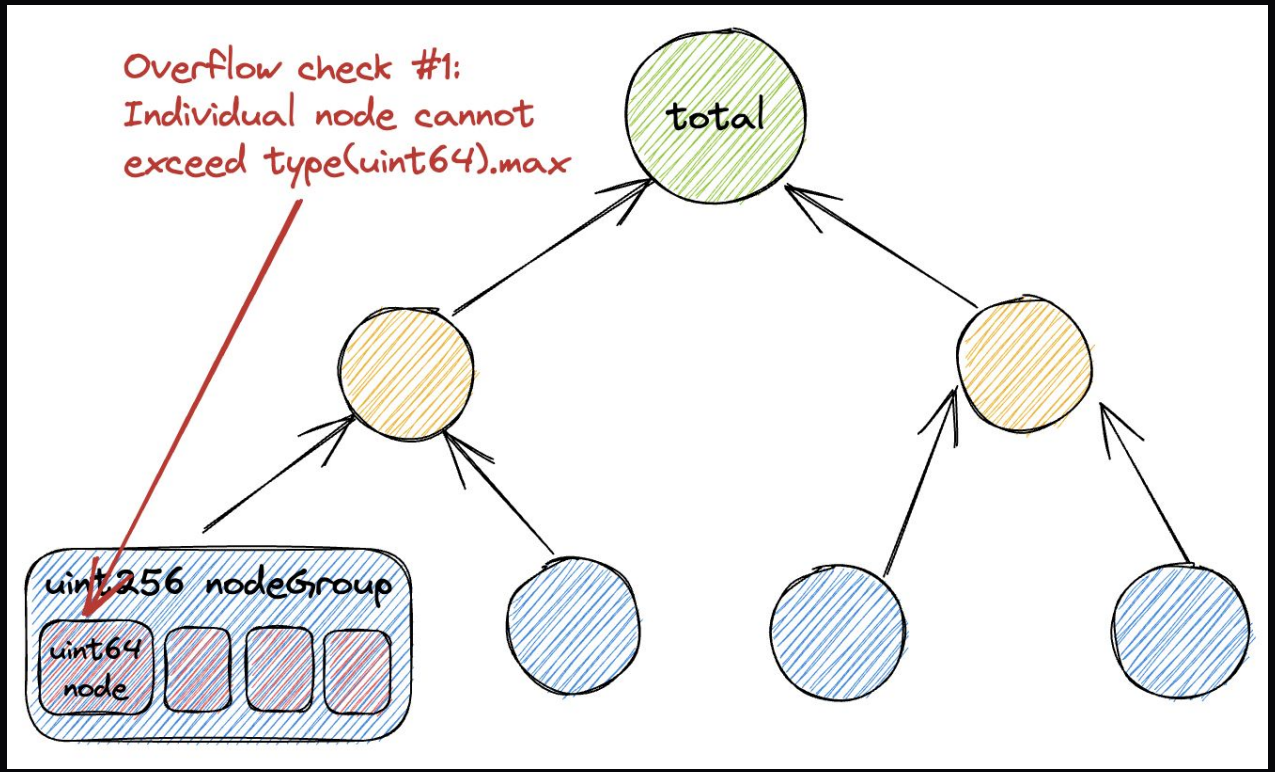
Addition overflow in Segmented Segment Tree

Crit



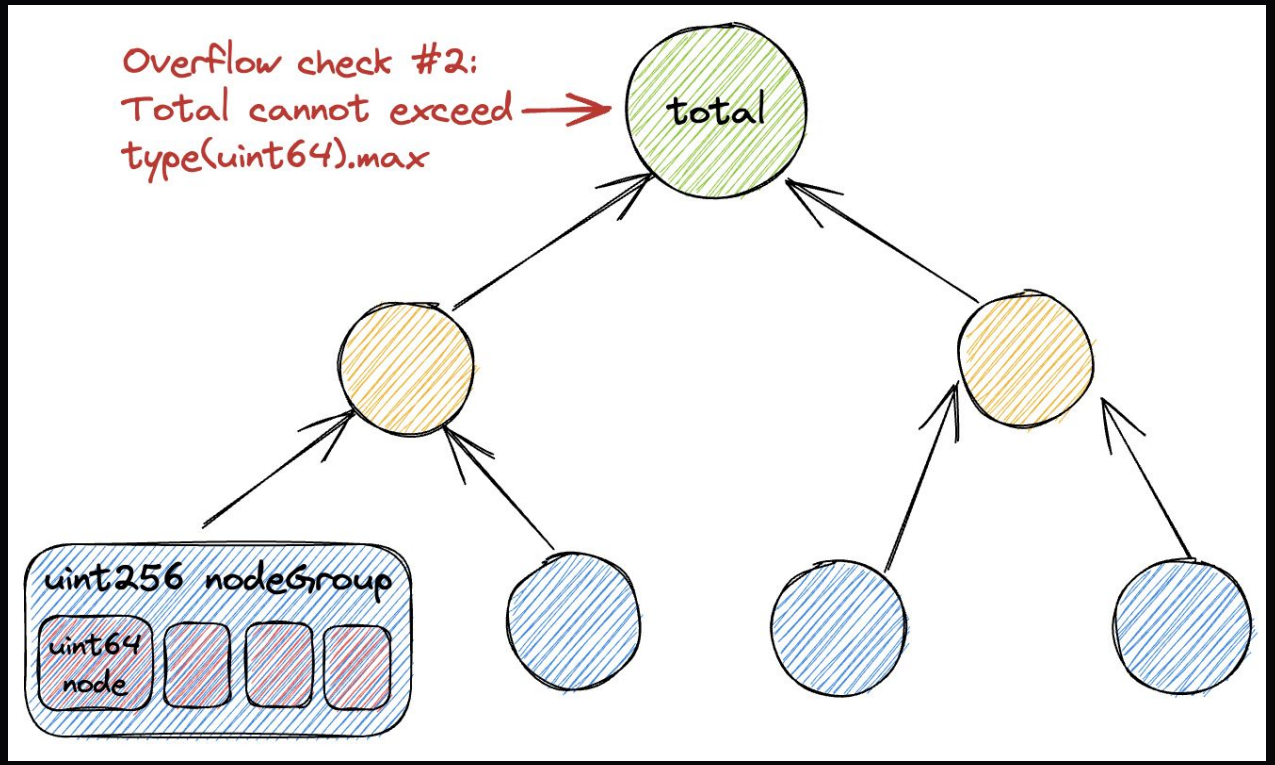
Addition overflow in Segmented Segment Tree

Crit



Addition overflow in Segmented Segment Tree

Crit



Addition overflow in Segmented Segment Tree

Crit

```
for (uint256 l = 0; l < _L; ++l) {
  LayerIndex memory layerIndex = indices[l];
  uint256 node = core.layers[l][layerIndex.group];
  core.layers[l][layerIndex.group] = node.update64(
      layerIndex.node,
      node.get64(layerIndex.node).addClean(diff)
  );
  if (total(core) > type(uint64).max) {
    revert SegmentedSegmentTree464Error(_TREE_MAX_ERROR);
  }
}
```

Overflow check #1

Overflow check #2

Addition overflow in Segmented Segment Tree

Crit

```
function addClean(uint64 current, uint64 cleanUint) internal pure returns (uint64) {
  assembly {
      current := add(add(current, iszero(current)), cleanUint)
  }
  if (current < cleanUint) {
      revert DirtyUint64Error(_OVERFLOW_ERROR);
  }
  return current;
}
```

Overflow check #1

Addition overflow in Segmented Segment Tree

Crit

```
function total(Core storage core) internal view returns (uint64) {
    return DirtyUint64.sumPackedUnsafe(core.layers[0][0], 0, _C)
        + DirtyUint64.sumPackedUnsafe(core.layers[0][1], 0, _C);
}
```

Addition overflow in Segmented Segment Tree

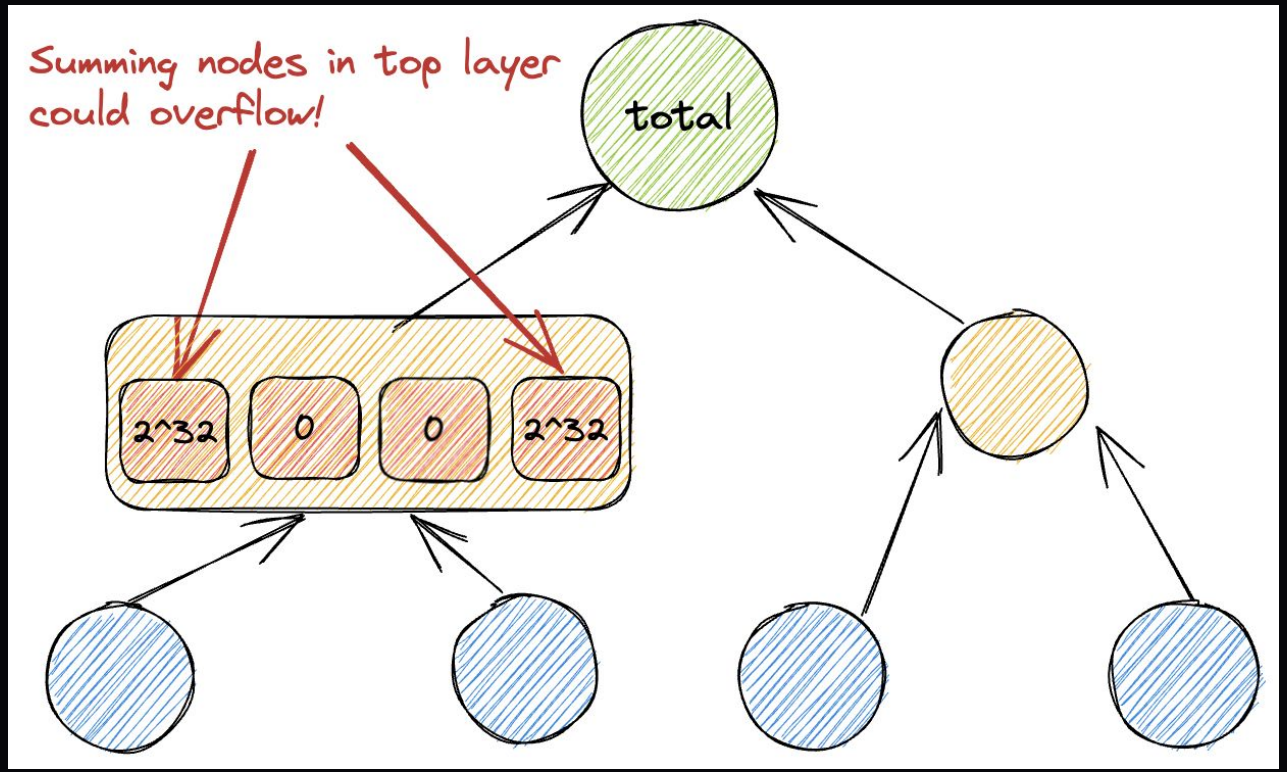
Crit

Uh oh… what if overflow happened here? Then overflow check #2 wouldn’t catch it because it already happened!

```
function total(Core storage core) internal view returns (uint64) {
    return DirtyUint64.sumPackedUnsafe(core.layers[0][0], 0, _C)
        + DirtyUint64.sumPackedUnsafe(core.layers[0][1], 0, _C);
}
```

Addition overflow in Segmented Segment Tree

Crit



Addition overflow in Segmented Segment Tree

Crit

```
function testTotalOverflow() public {
    uint64 half64 = type(uint64).max / 2 + 1;
    testWrapper.update(0, half64);
    // map to the right node of layer 0, group 0
    testWrapper.update(_MAX_ORDER / 2 - 1, half64);
    assertEq(testWrapper.total(), 0);
}
```

Takeaways

- Scrutinise each check performed
    - Are they done in the right place?
    - Can they be bypassed?
- Could the effect have already taken place before the check is performed?
    - Overflow already occurred, making the check “useless”

OrderBook DoS with blacklistable tokens (eg.USDC)

Crit

- Order queue at a price index were implemented as a ring buffer, up to 2^{15} orders
- Can only replace fulfilled orders (fully claimable) once MAX_ORDERS is reached
- Will send funds to the replaced order
- Can DoS if orderNFT is sent to a blacklisted address

Takeaways

Pull vs push fund transfers

OrderNFT theft due to ambiguous tokenId encoding / decoding scheme

Crit

```
function decodeId(uint256 id) public pure returns (CloberOrderBook.OrderKey memory) {
    uint8 isBid;
    uint16 priceIndex;
    uint232 orderIndex;
    assembly {
        orderIndex := id
        priceIndex := shr(232, id)
        isBid := shr(248, id)
    }
    return CloberOrderBook.OrderKey({isBid: isBid == 1, priceIndex: priceIndex, orderIndex: orderIndex});
}
```

OrderNFT theft due to ambiguous tokenId encoding / decoding scheme

Crit

Compression issue: leftmost 8 bits reduced to a bit comparison

0x_0000_0001... = bid, every other representation is treated as ask!

```
function decodeId(uint256 id) public pure returns (CloberOrderBook.OrderKey memory) {
    uint8 isBid;
    uint16 priceIndex;
    uint232 orderIndex;
    assembly {
        orderIndex := id
        priceIndex := shr(232, id)
        isBid := shr(248, id)
    }
    return CloberOrderBook.OrderKey({isBid: isBid == 1, priceIndex: priceIndex, orderIndex: orderIndex});
}
```

OrderNFT theft due to ambiguous tokenId encoding / decoding scheme

Crit

```
function ownerOf(uint256 tokenId) public view returns (address) {
    // tokenId of X + (1 << 255) decodes to tokenId X
    address owner = CloberOrderBook(market).getOrder(decodeId(tokenId)).owner;
    require(owner != address(0), Errors.ACCESS);
    return owner;
}
```

OrderNFT theft due to ambiguous tokenId encoding / decoding scheme

Crit

```
uint256 tokenId = orderToken.encodeId(order_key);
uint256 collidingTokenId = tokenId + (1 << 255);

// Attacker gives approval to victim (for trading) & another account to steal back NFT
vm.startPrank(attacker);
orderToken.approve(victim, tokenId);
orderToken.approve(attacker2, collidingTokenId);
vm.stopPrank();

// Victim transfers the NFT to themselves. (Or attacker trades it)
vm.startPrank(victim);
orderToken.transferFrom(attacker, victim, tokenId);
vm.stopPrank();

// Attacker steals the NFT
vm.startPrank(attacker2);
orderToken.transferFrom(victim, attacker2, collidingTokenId);
vm.stopPrank();
```

OrderNFT theft due to ambiguous tokenId encoding / decoding scheme

Crit

```
function decode(uint256 id) internal pure returns (OrderKey memory) {
    uint8 isBid;
    uint16 priceIndex;
    uint232 orderIndex;
    assembly {
        orderIndex := id
        priceIndex := shr(232, id)
        isBid := shr(248, id)
    }
    if (isBid > 1) {
        revert Errors.CloberError(Errors.INVALID_ID);
    }
    return OrderKey({isBid: isBid == 1, priceIndex: priceIndex, orderIndex: orderIndex});
}
```

Takeaways

- What are the consequences of unused / discarded / redundant bits?
- Are they safely ignored? Or can they somehow be maliciously used?
- Value collisions

OrderNFT theft due to controlling future and past tokens of same order index

Crit

- Order queue at a price index were implemented as a ring buffer, up to 2^{15} orders
- Modulo arithmetic:
    - X % N maps numbers to [0, N-1]
    - Achieve same effect with X & (N-1) where N - 1= 2^{y} - 1 (essentially extracting the last *y* bits)

OrderNFT theft due to controlling future and past tokens of same order index

Crit

```
function ownerOf(uint256 tokenId) public view returns (address) {
    // tokenId of X + (1 << 255) decodes to tokenId X
    address owner = CloberOrderBook(market).getOrder(decodeId(tokenId)).owner;
    require(owner != address(0), Errors.ACCESS);
    return owner;
}
```

```
function getOrder(OrderKey calldata orderKey) external view returns (Order memory) {
    return _getOrder(orderKey);
}
```

```
function _getOrder(OrderKey calldata orderKey) internal view returns (Order storage) {
    return _getQueue(orderKey.isBid, orderKey.priceIndex).orders[orderKey.orderIndex & _MAX_ORDER_M];
}
```

OrderNFT theft due to controlling future and past tokens of same order index

Crit

```
function ownerOf(uint256 tokenId) public view returns (address) {
    // tokenId of X + (1 << 255) decodes to tokenId X
    address owner = CloberOrderBook(market).getOrder(decodeId(tokenId)).owner;
    require(owner != address(0), Errors.ACCESS);
    return owner;
}
```

```
function getOrder(OrderKey calldata orderKey) external view returns (Order memory) {
    return _getOrder(orderKey);
}
```

```
function _getOrder(OrderKey calldata orderKey) internal view returns (Order storage) {
    return _getQueue(orderKey.isBid, orderKey.priceIndex).orders[orderKey.orderIndex & _MAX_ORDER_M];
}
```

OrderNFT theft due to controlling future and past tokens of same order index

Crit

What’s the problem with this?

```
function _getOrder(OrderKey calldata orderKey) internal view returns (Order storage) {
    return _getQueue(orderKey.isBid, orderKey.priceIndex).orders[orderKey.orderIndex & _MAX_ORDER_M];
}
```

OrderNFT theft due to controlling future and past tokens of same order index

Crit

Existing owner is the owner of future (& past) looped orders that map to the same index!

Eg. orderIndex 0's owner is also owner of orderIndex 2^{15}, … , k * 2^{15}

```
function _getOrder(OrderKey calldata orderKey) internal view returns (Order storage) {
    return _getQueue(orderKey.isBid, orderKey.priceIndex).orders[orderKey.orderIndex & _MAX_ORDER_M];
}
```

OrderNFT theft due to controlling future and past tokens of same order index

Crit

An attacker can set approvals of future token IDs to himself, then steal future minted NFTs!

```
function _getOrder(OrderKey calldata orderKey) internal view returns (Order storage) {
    return _getQueue(orderKey.isBid, orderKey.priceIndex).orders[orderKey.orderIndex & _MAX_ORDER_M];
}
```

Takeaways

- Thinking bi-directional
- Collisions with modulo arithmetic: What are the consequences of numbers wrapping around?
  - Applies to ZK because of finite prime fields

Takeaways

Aztech 2.0 Bug: Pedersen hash input checks

The bug was that, when validating the sum of the windows equalled the input field element, we were validating this [*mod p*], where *p* is the native circuit modulus.

This meant that every hash input effectively had two possible representations in 2-bit window form (the actual binary value or the value + [*p*]). This meant that every Pedersen hash effectively had two different outputs.

A consequence of this bug is that it was possible to generate two nullifiers for every note. This would enable a double-spending attack.

P.S. check out HickupHH3's Twitter pinned tweet for more ZK resources

Rounding up of taker fees of constituent orders may exceed collected fee

High

Filling order:

```
outputAmount -= _calculateTakerFeeAmount(outputAmount, true);
```

Claiming filled order:

```
takerFeeAmount = _calculateTakerFeeAmount(takeAmount, true);
```

```
function _calculateTakerFeeAmount(uint256 takeAmount, bool roundingUp) internal view returns (uint256) {
    // takerFee is always positive
    return Math.divide(takeAmount * takerFee, _FEE_PRECISION, roundingUp);
}
```

- Fees taken from filler all goes to the makers
- outputAmount is the full taken amount, while takeAmount is the individual filled maker order

Rounding up of taker fees of constituent orders may exceed collected fee

High

Was it possible that the fees collected will exceed the fees to be claimed due to rounding?

Rounding up of taker fees of constituent orders may exceed collected fee

High

takerFee = 100_011 (10.0011%)

Maker orders of amounts 400_000 & 377_000

Total amount = 400_000 + 377_000 = 777_000

Rounding up of taker fees of constituent orders may exceed collected fee

High

takerFee for filling both orders

= 777_000 * 100_011 / 1_000_000 = 77_708.547

= 77_709

Maker fees:

400_000 * 100_011 / 1_000_000 = 37705

377_000 * 100_011 / 1_000_000 = 40005

Total maker fees = 37705 + 40005 = 77_710 > 77_709

Rounding up of taker fees of constituent orders may exceed collected fee

High

Fix: fees rounded down for claim calculations

```
// rounding down to prevent insufficient balance
takerFeeAmount = _calculateTakerFeeAmount(takeAmount, false);
```

Takeaways

Think about rounding issues when division is performed across segments:

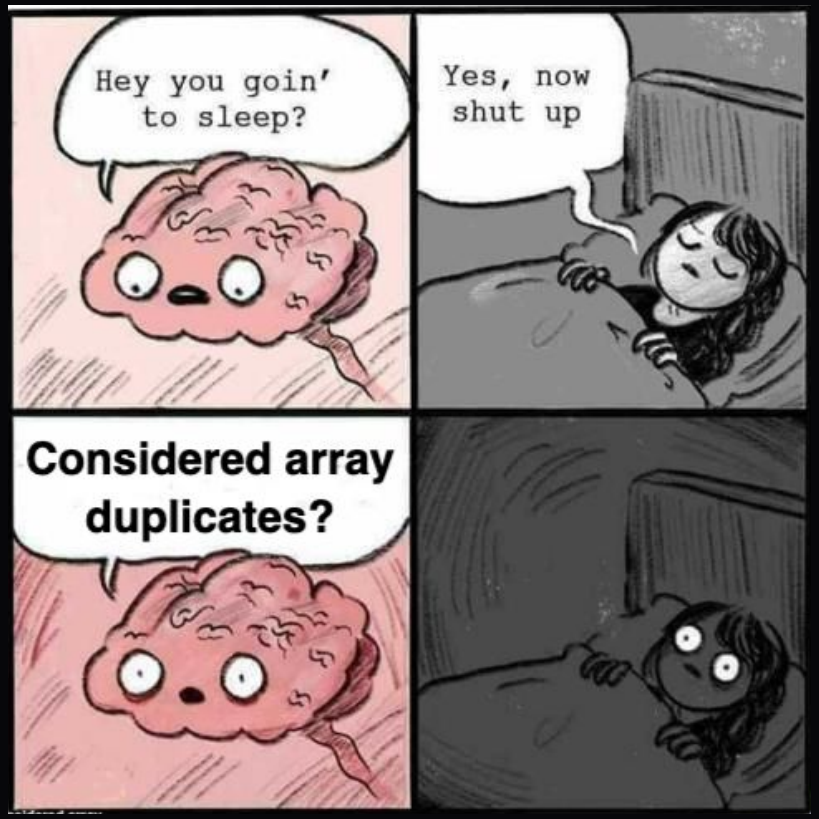
21/3 != 7/3 + 7/3 + 7/3

Takeaways

Build up your (mental) model asap

=> passive but intentional thinking / letting your mind wander

Takeaways


Hey you goin' to sleep?
Yes, now shut up
Considered array duplicates?

Takeaways

Complete the thought process

- If what you found wasn’t a bug, note down the reasoning (question / challenge it another time)
- Can the problem exist in another part of the codebase in a different form?

marketOrder() with expendOutput reverts with SlippageError with max tolerance

High

```
if (!expendInput) {
    // Increase requestedAmount by fee when expendInput is false
    requestedAmount = _calculateTakeAmountBeforeFees(requestedAmount);
}
```

```
while (word > 0) {
  if (limitPriceIndex < currentIndex) break;
  if (isTakingBidSide) currentIndex = ~currentIndex;

  (uint256 _inputAmount, uint256 _outputAmount, ) = _expectTake(
      isTakingBidSide,
      requestedAmount,
      currentIndex,
      expendInput
  );
  inputAmount += _inputAmount;
  outputAmount += _outputAmount - _calculateTakerFeeAmount(outputAmount, true);
```

Order owner isn't zeroed after burning

Med

```
function _burnToken(uint256 orderId) internal {
  // owner isn't zeroed-out, ownerOf() will return current owner
  CloberOrderNFT(orderToken).onBurn(orderId);
}
```

Decoupled NFT & Market Ownership

Low

- Market host (address stored in factory) entitled to 80% of fees collected
- OrderNFT owner can set URI (inherits Ownable)
- Both point to the same address
- However, 1 could transfer the market host or orderNFT owner without the other

Decoupled NFT & Market Ownership

```
function owner() external view returns (address) {
    return _getHost();
}

function _getHost() internal view returns (address) {
    return _factory.getMarketHost(market);
}
```

Takeaways

- It can be easy to spot what’s wrong with present code, but.. it’s tougher to spot what’s absent

Takeaways

- It can be easy to spot what’s wrong with present code, but.. it’s tougher to spot what’s absent

Takeaways

- It can be easy to spot what's wrong with present code, but.. it's tougher to spot what's absent
  - State handling: is there anything that should've been updated, but isn't?
- Especially when NFT ownership / tokens are transferred
  - Eg. liquidity mining reward accounting

Other Takeaways

- Scrutinise code changes
  - Couple of bugs found were introduced after audit started
- If in doubt regarding how code behaves, ping fellow members & clarify with client
  - Eg. Claim functionality was designed to support 3rd party operators, should skip instead of reverting

Other Takeaways

Communication

Questions?