SHERLOCK

SHERLOCK SECURITY REVIEW FOR

KNOX

Prepared for:	Knox
Prepared by:	Sherlock
Lead Security Expert:	ArbitraryExecution
Dates Audited:	September 29 - October 13, 2022
Prepared on:	October 31, 2022

Introduction

Knox Finance is a DeFi options platform focused on providing predictable yields and risk management in the form of structured products.

Scope

Commit Hash: b0a872d25caeb833bab17e69ef0de51d7ca862a2

```
- contracts/auction/
  - Auction.sol
  - AuctionInternal.sol
  - AuctionProxy.sol
  - AuctionStorage.sol
  - OrderBook.sol

- contracts/libraries/
  - OptionMath.sol

- contracts/pricer/
  - Pricer.sol
  - PricerInternal.sol

- contracts/queue/
  - Queue.sol
  - QueueInternal.sol
  - QueueProxy.sol
  - QueueStorage.sol

- contracts/vault/
  - VaultAdmin.sol
  - VaultBase.sol
  - VaultDiamond.sol
  - VaultInternal.sol
  - VaultStorage.sol
  - VaultView.sol
```

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Total Issues

Medium	High
7	4

Security Experts

Trumpero
yixxas
hansfriese
ctf_sec
__141345__
0x52
ArbitraryExecution
0xNazgul
dipp
shung
bin2chen
rvierdiiev
joestakey
Olivierdem
ak1
csanuragjain
minhquanym
berndartmueller
ali_shehab
Jeiwan
jayphbee
GalloDaSballo
cccz
Ruhum

Issue H-1: Underflow in `_previewWithdraw` could prevent withdrawals

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/106

Found by

dipp, __141345__, Trumpero, 0x52, hansfriese, yixxas

Summary

An underflow in the `_previewWithdraw` function in `AuctionInternal.sol` due to totalContractsSold exceeding auction.totalContracts could prevent users from withdrawing options.

Vulnerability Detail

The `_previewWithdraw` function returns the fill and refund amounts for a buyer by looping over all orders. A totalContractsSold variable is used to track the amount of contracts sold as the loop iterates over all orders. If the current order's size + totalContractsSold exceeds the auction's totalContracts then the order will only be filled partially. The calculation for the partial fill (remainder) is given on line 318. This will lead to an underflow if totalContractsSold > the auction's totalContracts which would happen if there are multiple orders that cause the totalContractsSold variable to exceed totalContracts.

The totalContractsSold variable in `_previewWithdraw` could exceed the auction.totalContracts due to the contracts sold before the start of an auction through limit orders not being limited. When an order is added, _finalizeAuction is only called if the auction has started. The `_finalizeAuction` function will call the `_processOrders` function which will return true if the auction has reached 100% utilization. Since limit orders can be made before the start of an auction, _finalizeAuction is not called and any amount of new orders may be made.

Example: The buyer makes a limit order with size > auction.totalContracts. They then make another order with size of anything. These orders are made before the start of the auction so `_processOrders` is not called for every new order and totalContractsSold can exceed totalContracts. When `_previewWithdraw` is called, after the buyer's first order is processed, totalContractsSold > auction.totalContracts so the condition on line 313 passes. Since totalContractsSold > auction.totalContracts the calculation on line 318 underflows and the transaction reverts. The `_previewWithdraw` function and thus the `_withdraw` function is uncallable.

Test code added to `Auction.behaviour.ts`, under the `addLimitOrder(uint64,int128,uint256)` section:

```
it("previewWithdraw reverts if buyer has too many contracts", async () => {
      assert.isEmpty(await auction.getEpochsByBuyer(addresses.buyer1));

      await asset
        .connect(signers.buyer1)
        .approve(addresses.auction, ethers.constants.MaxUint256);

      const totalContracts = await auction.getTotalContracts(epoch);
      await auction.addLimitOrder(
        epoch,
        fixedFromFloat(params.price.max),
        totalContracts.mul(2)
      );

      await auction.addLimitOrder(
        epoch,
        fixedFromFloat(params.price.max),
        totalContracts.div(2)
      );

      const epochByBuyer = await auction.getEpochsByBuyer(addresses.buyer1);

      assert.equal(epochByBuyer.length, 1);
      assert.bnEqual(epochByBuyer[0], epoch);

      await expect(auction.callStatic[
        "previewWithdraw(uint64)"
      ](epoch)).to.be.reverted;
    });
```

The test code above shows a buyer is able to add an order with size auction.totalContracts*2 and a subsequent order with size auction.totalContracts/2. The `previewWithdraw` function reverts when called.

Impact

Users would be unable to withdraw from the Auction contract.

Code Snippet

AuctionInternal.sol:_previewWithdraw#L312-L321

```
if (
    totalContractsSold + data.size >= auction.totalContracts
) {
```

```
    // if part of the current order exceeds the total contracts available,
↪   partially
    // fill the order, and refund the remainder
    uint256 remainder =
        auction.totalContracts - totalContractsSold;

    cost = lastPrice64x64.mulu(remainder);
    fill += remainder;
```

AuctionInternal.sol:_validateLimitOrder#L479-L489

```
function _validateLimitOrder(
    AuctionStorage.Layout storage l,
    int128 price64x64,
    uint256 size
) internal view returns (uint256) {
    require(price64x64 > 0, "price <= 0");
    require(size >= l.minSize, "size < minimum");

    uint256 cost = price64x64.mulu(size);
    return cost;
}
```

AuctionInternal.sol:_addOrder#L545-L562

```
function _addOrder(
    AuctionStorage.Layout storage l,
    AuctionStorage.Auction storage auction,
    uint64 epoch,
    int128 price64x64,
    uint256 size,
    bool isLimitOrder
) internal {
    l.epochsByBuyer[msg.sender].add(epoch);

    uint256 id = l.orderbooks[epoch]._insert(price64x64, size, msg.sender);

    if (block.timestamp >= auction.startTime) {
        _finalizeAuction(l, auction, epoch);
    }

    emit OrderAdded(epoch, id, msg.sender, price64x64, size, isLimitOrder);
}
```

Tool used

Manual Review

Recommendation

The loop in `_previewWithdraw` should check if the current totalContractsSold is >= totalContracts. If it is then the remainder should be set to 0 which would allow the current order to be fully refunded.

Additionally, the orders for an auction should be checked before the auction starts. In `_addOrder`, consider adding a condition that will call `_processOrders` if the auction has not started yet. If `_processOrders` returns true then do not allow the order to be added. Or just allow the auction to be finalized before it starts if the total contracts sold has reached the auction's totalContracts.

Issue H-2: [NAZ-M6] Unbounded loop in `_previewWithdraw()_redeemMax()` Can Lead To DoS

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/102

Found by

__141345__, 0xNazgul, ctf_sec

Summary

There are some unbounded loops that can lead to DoS.

Vulnerability Detail

The loop inside of `_previewWithdraw()` goes through all orders in `orderbook` and checks if the `data.buyer` is the `buyer` passed in parameter. It does some checks, math and an external call to remove the `data.id` from the orderbook. With all of this happening in the loop and costing gas it may revert due to exceeding the block size gas limit. This is the same case for `_redeemMax()` but has more gas costly executions with `transfer()` being involved.

Impact

There are over thousands of orders that the loop has to go through, along with the massive amount of orders that are the `buyers`. Half way through the execution fails due to exceeding the block size gas limit.

Code Snippet

AuctionInternal.solL298, QueueInternal.solL140

Tool used

Manual Review

Recommendation

Consider avoiding all the actions executed in a single transaction, especially when calls are executed as part of a loop.

Discussion

0xCourtney

It might make sense to treat `redeemMax` as a separate issue since it's in a different contract and therefore relies on a different mechanism than the other functions mentioned. `redeemMax` implements a for-loop that iterates through all of the tokenId's held by an account. These tokenId's accumulate whenever a user deposits collateral into the queue during different epochs. The `redeemMax` function is called every time a user deposits, into the Queue or withdraws from the Vault. This makes it impossible to accumulate more than one new tokenId at a time.

This applies specifically to issues #15, and #102.

Evert0x

@0xCourtney would you consider 15 / 102 a different severity than the others? Based on your comment I assume this issue related to `redeemMax` is invalid but the others (24, 25, 82, 85) are valid. Is that correct?

0xCourtney

#25 is actually incorrect. We do check that size > minSize for the auction orders for _validateLimitOrder and _validateMarketOrder #102 should be two separate issues, one for `previewWithdraw` another for `redeemMax`. The issue mentioned about `redeemMax` should be low or informational for the reasons stated above. #15 likewise should be low or informational as its related to `redeemMax` #24, #82, and #85 are issues we plan to fix.

Evert0x

For #25 `minSize` is set in the `AuctionProxy` constructor but no `>0` check is done, but we consider that a low issue.

Evert0x

Just for confirmation I kept #102 the main issue, although it's referencing two separate issues. #24, #82 and #85 are duplicates of this. Rest of the issues are low/invalid

Issue H-3: Auction can potentially sell more contracts than it has collateral for.

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/80

Found by

hansfriese, yixxas

Summary

`auction.totalContracts` is determined by the amount of collateral the protocol has received. After an auction has ended, users are allowed to withdraw and what they receive depends on whether their orders have filled, or they receive a refund, or a mixture of both. However, wrong accounting in `_previewWithdraw()` can lead to the `fill` and `refund` value to be calculated wrongly.

Vulnerability Detail

Each time a withdrawal is made, the order is removed from the order book as seen in L339 in `_previewWithdraw()`. Now, the issue here is in the line `totalContractsSold +=data.size`. If a user with an order that is higher priced, that is with a lower index in the order book(since order book is in decreasing order based on price), chooses to withdraw first, their order is removed from the order book. Now, the next user who does a withdraw will call this same function, but `totalContractsSold` is calculated from 0 again. This leads to problems and I illustrate with a simple example below.

Assume, `auction.totalContracts=10`

Alice first `addLimitOrder()` with `price=10`, `size=10`. Bob then `addLimitOrder()` with `price=10`, 'size = 1'.

Now order book have 2 orders.

`processOrder()` is then called, since utilisation reaches a 100%, clearing price is set to 10 and all contracts are sold to Alice. Now, Alice does a withdraw first and withdraws successfully with `fill=10,refund=0`. Now Bob tries to withdraw, the previous order is removed from the order book, so when `i=1`, it enters the `if(data.price64x64>=lastPrice64x64)` check. This time, `if(totalContractsSold+data.size>=auction.totalContracts)` check does not pass since `totalContractsSold=0,data.size=1,auction.totalContracts=10`. It will then enter the else part where `fill+=data.size`. This means that additional contracts are being sold to Bob even though it has been previously sold to Alice which exceeds the limit that the collateral allows.

Impact

Auction is selling more contracts than it has collateral for which creates plenty of liquidity problems / issues with risk as options are now "naked".

Code Snippet

AuctionInternal.sol#L279-L347

```
function _previewWithdraw(
    AuctionStorage.Layout storage l,
    bool isPreview,
    uint64 epoch,
    address buyer
) private returns (uint256, uint256) {
    ...
    uint256 totalContractsSold;
    ...
    // traverse the order book and return orders placed by the buyer
    for (uint256 i = 1; i <= length; i++) {
        OrderBook.Data memory data = orderbook._getOrderById(next);
        next = orderbook._getNextOrder(next);

        if (data.buyer == buyer) {
            if (
                lastPrice64x64 < type(int128).max &&
                data.price64x64 >= lastPrice64x64
            ) {
                // if the auction has not been cancelled, and the order price is
↪   greater than or
                // equal to the last price, fill the order and calculate the
↪   refund amount
                uint256 paid = data.price64x64.mulu(data.size);
                uint256 cost = lastPrice64x64.mulu(data.size);

                if (
                    totalContractsSold + data.size >= auction.totalContracts
                ) {
                    // if part of the current order exceeds the total contracts
↪   available, partially
                    // fill the order, and refund the remainder
                    uint256 remainder =
                        auction.totalContracts - totalContractsSold;

                    cost = lastPrice64x64.mulu(remainder);
                    fill += remainder;
                } else {
                    // otherwise, fill the entire order
```

```
                    fill += data.size;
                }

                // the refund takes the difference between the amount paid and
↪   the "true" cost of
                // of the order. the "true" cost can be calculated when the
↪   clearing price has been
                // set.
                refund += paid - cost;
            } else {
                // if last price >= type(int128).max, auction has been cancelled,
↪   only send refund
                // if price < last price, the bid is too low, only send refund
                refund += data.price64x64.mulu(data.size);
            }

            if (!isPreview) {
                // when a withdrawal is made, remove the order from the order
↪   book
                orderbook._remove(data.id);
            }
        }

        totalContractsSold += data.size;
    }

    return (refund, fill);
}
```

Tool used

Manual Review

Recommendation

This problem arises due to how orders that have the same price as the clearing price, yet should not be filled due to exceeding the limit is not accounted for. A check in `_previewWithdraw()` needs to be done to prevent this edge case.

I believe removing an order from the order book after withdrawal is done to prevent multiple withdrawals from the same user. If this is the case, we can use a mapping to check this instead, so that `totalContractsSold` remains accurate. We can then refund users once this exceeds `auction.totalContracts`.

Issue H-4: Wrong implementation of orderbook can make user can't get their fund back

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/66

Found by

Trumpero

Lines of code

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/auction/OrderBook.sol#L240 https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/auction/OrderBook.sol#L182-L190

Summary

When a user remove an order, next user call `addLimitOrder` can override the latest order with his/her order. It will make one who is owner of that latest order lose their fund.

Vulnerability Detail

Function `_remove` will decrease value of `index.length` by 1 when an order is removed

Instead of reserving `id` of removed order to reuse for next created order, function `_insert` use the id of new order is `index.length+1`

It will override the latest order with new order's data.

For example

- Alice create an order with price = 10 --> `id=1,index.length=1`
- Bob create an order with price = 20 --> `id=2,index.length=2`
- Alice cancel order `id=1` --> `index.length=1`
- Candice create new order with price = 30
  - At this time, new order will have `id=index.length+1=1+1=2`. It will override the state of Bob's order: price from 20 -> 30

Impact

User whose order is overrided can't withdraw their refund `ERC20` and their exercised tokens.

Code Snippet

To check with test, u can use this file https://gist.github.com/Trumpero/adbcd84c33f71856dbf379f581e8abbb I write one more describe `::Bug` beside your original describe `::Auction` in file `Auction.behavior.ts` (just too lazy to write a new one).

Tool used

Hardhat

Recommendation

Use an array to store unused (removed) id, then assign each id to the new limit order created instead of using `index.length`.

Issue M-1: ## Auction can be ended with large limit order

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/153

Found by

ArbitraryExecution

In the function AuctionInternal._previewWithdraw, the number type(int128).max is used as a sentinel value for deciding whether the auction is canceled. However, an order for all contracts at a price of type(int128).max passes all requires checks for limit orders.

If such an order is placed at the start of the auction, the auction can be finalized and the trader will be refunded the entire amount when the auction is processed or canceled. This effectively prevents the auction from taking place.

Recommendation Use AuctionStorage.Status instead of sentinel values for determining whether the auction is in the canceled state.

Issue M-2: Chainlink's `latestRoundData` might return stale or incorrect results

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/137

Found by

Jeiwan, csanuragjain, berndartmueller, jayphbee, joestakey, Olivierdem, Ruhum, GalloDaSballo, __141345__, Trumpero, ArbitraryExecution, hansfriese, ali_shehab, cccz, 0xNazgul, ak1, ctf_sec, minhquanym

Summary

Chainlink's `latestRoundData()` is used but there is no check if the return value indicates stale data. This could lead to stale prices according to the Chainlink documentation:

- https://docs.chain.link/docs/historical-price-data/#historical-rounds

Vulnerability Detail

The `PricerInternal._latestAnswer64x64` function uses Chainlink's `latestRoundData()` to get the latest price. However, there is no check if the return value indicates stale data.

Impact

The `PricerInternal` could return stale price data for the underlying asset.

Code Snippet

PricerInternal._latestAnswer64x64

```
/**
  * @notice gets the latest price of the underlying denominated in the base
  * @return price of underlying asset as 64x64 fixed point number
  */
function _latestAnswer64x64() internal view returns (int128) {
    (, int256 basePrice, , , ) = BaseSpotOracle.latestRoundData();
    (, int256 underlyingPrice, , , ) =
        UnderlyingSpotOracle.latestRoundData();

    return ABDKMath64x64.divi(underlyingPrice, basePrice);
}
```

Tool Used

Manual review

Recommendation

Consider adding checks for stale data. e.g

```
(uint80 roundId, int256 basePrice, , uint256 updatedAt, uint80 answeredInRound) =
↪  BaseSpotOracle.latestRoundData();

require(answeredInRound >= roundId, "Price stale");
require(block.timestamp - updatedAt < PRICE_ORACLE_STALE_THRESHOLD, "Price round
↪  incomplete");
```

Issue M-3: `epochsByBuyer[]` can lose records

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/86

Found by

rvierdiiev, hansfriese, __141345__, bin2chen

Summary

When `cancelLimitOrder()`, the `epoch` will be removed from `epochsByBuyer[]`, but the user could have other orders on the orderbook, the record will be inaccurate and mislead the users.

Vulnerability Detail

If a user put 2 limit orders with different prices, 2 separate id will be assigned, but only 1 epoch will be added to the `epochsByBuyer[]` array. If the user cancel 1 of the orders, the `epoch` will be removed from `epochsByBuyer[]`, it will be impossible to track the other orders put by the user. And the `getEpochsByBuyer()` function will return inaccurate result.

Impact

If some users rely on the results of `getEpochsByBuyer()` for new orders, the returned inaccurate results could be misleading, and cause potential loss to the users due to wrong information.

Code Snippet

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/auction/AuctionInternal.sol#L545-L553

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/auction/Auction.sol#L220

Tool used

Manual Review

Recommendation

Only remove the `epochsByBuyer` records when all the orders of the user is cancelled.

Issue M-4: Users can avoid performance fees by withdrawing before the end of the epoch forcing other users to pay their fees

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/75

Found by

0x52

Summary

No performance fees are taken when user withdraws early from the vault but their withdrawal value will be used to take fees, which will be taken from other users.

Vulnerability Detail

```
uint256 adjustedTotalAssets = _totalAssets() + l.totalWithdrawals;

if (adjustedTotalAssets > l.lastTotalAssets) {
    netIncome = adjustedTotalAssets - l.lastTotalAssets;

    feeInCollateral = l.performanceFee64x64.mulu(netIncome);

    ERC20.safeTransfer(l.feeRecipient, feeInCollateral);
}
```

When taking the performance fees, it factors in both the current assets of the vault as well as the total value of withdrawals that happened during the epoch. Fees are paid from the collateral tokens in the vault, at the end of the epoch. Paying the fees like this reduces the share price of all users, which effectively works as a fee applied to all users. The problem is that withdraws that take place during the epoch are not subject to this fee and the total value of all their withdrawals are added to the adjusted assets of the vault. This means that they don't pay any performance fee but the fee is still taken from the vault collateral. In effect they completely avoid the fee force all there other users of the vault to pay it for them.

Impact

User can avoid performance fees and force other users to pay them

Code Snippet

VaultInternal.sol#L504-L532

Tool used

Manual Review

Recommendation

Fees should be taken on withdrawals that occur before vault is settled

Issue M-5: _getNextFriday() returns wrong value when timestamp is between Monday 12am and 8am.

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/47

Found by

yixxas

Summary

`_getNextFriday(Monday)` should return next Friday 8am timestamp. But if timestamp is on a Monday 12am - 8am, it will wrongly return the same week Friday.

Vulnerability Detail

`friday8am-timestamp<4days` check fails when the `timestamp` is between Monday 12am to Monday 8am, since `friday8am-timestamp>4days`. It will wrongly return the same week Friday instead of next Friday.

Impact

`_setOptionParameters()` relies on `_getNextFriday()` to set option's expiry. When the timestamp is between Monday 12am and Monday 8am, expiry date will be wrong causing malfunction of protocol.

Code Snippet

VaultInternal.sol#L746-L759

```
    function _getNextFriday(uint256 timestamp) internal pure returns (uint256) {
        // dayOfWeek = 0 (sunday) - 6 (saturday)
        uint256 dayOfWeek = ((timestamp / 1 days) + 4) % 7;
        uint256 nextFriday = timestamp + ((7 + 5 - dayOfWeek) % 7) * 1 days;
        uint256 friday8am = nextFriday - (nextFriday % (24 hours)) + (8 hours);

        // If the timestamp is on a Friday or between Monday-Thursday
        // return Friday of the following week
        if (timestamp >= friday8am || friday8am - timestamp < 4 days) {
            friday8am += 7 days;
        }
        return friday8am;
    }
}
```

Tool used

Manual Review

Recommendation

Add 8 hours to the check when doing subtraction of `friday8am-timestamp`.

```
    function _getNextFriday(uint256 timestamp) internal pure returns (uint256) {
        // dayOfWeek = 0 (sunday) - 6 (saturday)
        uint256 dayOfWeek = ((timestamp / 1 days) + 4) % 7;
        uint256 nextFriday = timestamp + ((7 + 5 - dayOfWeek) % 7) * 1 days;
        uint256 friday8am = nextFriday - (nextFriday % (24 hours)) + (8 hours);

        // If the timestamp is on a Friday or between Monday-Thursday
        // return Friday of the following week
-       if (timestamp >= friday8am || friday8am - timestamp < 4 days) {
+       if (timestamp >= friday8am || friday8am - timestamp < (4 days + 8 hours))
 ↪  {
            friday8am += 7 days;
        }
        return friday8am;
    }
}
```

Issue M-6: Internal `OptionMath._getPositivePlaceValues()` function do not handle values below `185`

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/43

Found by

shung, ArbitraryExecution

Summary

Internal `OptionMath._getPositivePlaceValues()` function do not handle values below `185`.

Vulnerability Detail

`OptionMath._getPositivePlaceValues()` is a library function used by special floor and ceiling functions which are in turn used in the calculation of the strike price. However, `_getPositivePlaceValues()` function incorrectly reverts when the provided value is below `185`. This happens because a 64x64 `185` equals to a wad value with one digit, which is represented internally with one extra decimal. Since the division by 100 will return zero for a number with less than three digits, the division in the following line reverts due to division by zero.

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/libraries/OptionMath.sol#L101-L103

Impact

This might prevent auctions from starting due to the following execution flow `VaultAdmin.initializeAuction()->VaultInternal._setOptionParameters()->Pricer.snapToGrid64x64()->OptionMath.ceil64x64()->OptionMath._getPositivePlaceValues()`.

There can also be more serious issues if this library is reused in other places.

Code Snippet

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/libraries/OptionMath.sol#L93-L103

Tool used

Manual Review

Recommendation

Applying the following diff will properly handle division by zero, such that values below 185 can still be rounded up or down.

```
diff --git a/knox-contracts/contracts/libraries/OptionMath.sol
 ↪  b/knox-contracts/contracts/libraries/OptionMath.sol
index 746dd78..8ed2bbc 100644
--- a/knox-contracts/contracts/libraries/OptionMath.sol
+++ b/knox-contracts/contracts/libraries/OptionMath.sol
@@ -92,15 +92,21 @@ library OptionMath {

         // setup the first place value
         values[0].ruler = ruler;
-        values[0].value = (integer / values[0].ruler) % 10;
-
-        // setup the second place value
-        values[1].ruler = ruler / 10;
-        values[1].value = (integer / values[1].ruler) % 10;
-
-        // setup the third place value
-        values[2].ruler = ruler / 100;
-        values[2].value = (integer / values[2].ruler) % 10;
+        if (values[0].ruler != 0) {
+            values[0].value = (integer / values[0].ruler) % 10;
+
+            // setup the second place value
+            values[1].ruler = ruler / 10;
+            if (values[1].ruler != 0) {
+                values[1].value = (integer / values[1].ruler) % 10;
+
+                // setup the third place value
+                values[2].ruler = ruler / 100;
+                if (values[2].ruler != 0) {
+                    values[2].value = (integer / values[2].ruler) % 10;
+                }
+            }
+        }

         return (integer, values);
     }
```

Issue M-7: processAuction() in VaultAdmin.sol can be called multiple times by keeper if the auction is canceled.

Source: https://github.com/sherlock-audit/2022-09-knox-judging/issues/26

Found by

ctf_sec

Summary

processAuction() in VaultAdmin.sol can be called multiple times by keeper if the auction is canceled.

Vulnerability Detail

processAuction() in VaultAdmin.sol can be called multiple times by keeper, the code below would execute more than one times if the auction is canceled.

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/vault/VaultAdmin.sol#L259-L280

because it is the line of code inside the function processAuction in VaultAdmin.sol below that can change the auction status to PROCESSED.

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/vault/VaultAdmin.sol#L326

this code only runs when the auction is finalized, it not finalized, the auction is in Canceled State and

```
bool cancelled = l.Auction.isCancelled(lastEpoch);
     bool finalized = l.Auction.isFinalized(lastEpoch);

     require(
         (!finalized && cancelled) || (finalized && !cancelled),
         "auction is not finalized nor cancelled"
     );
```

would always pass because the auction is in cancel state.

Impact

Why the processAuction should not be called multiple times?

In the first time it is called, the withdrawal lock is released so user can withdraw fund,

```
// deactivates withdrawal lock
 l.auctionProcessed = true;
```

then if we called again, the lastTotalAssets can be updated multiple times.

```
// stores the last total asset amount, this is effectively the amount of assets
↪  held
// in the vault at the start of the auction
l.lastTotalAssets = _totalAssets();
```

the total asset can be lower and lower because people are withdrawing their fund.

then when _collectPerformanceFee is called, the performance may still be collected

https://github.com/sherlock-audit/2022-09-knox/blob/main/knox-contracts/contracts/vault/VaultInternal.sol#L513-L530

Code Snippet

Tool used

Manual Review

Recommendation

We recommend the project lock the epoch and make it impossible for keeper to call the processAuction again.

Discussion

0xCourtney

The `Keeper` is an EOA owned/controlled by the protocol team and therefore considered trusted.

Evert0x

@0xCourtney as there are require statements based on the auction state, is it a valid use case that `processAuction()` get's called multiple times (by the keeper)? If not I can see the argument for the missing check.

0xCourtney

No, this function should only be called once. We'll add a guard to prevent multiple calls.