SHERLOCK

# SHERLOCK SECURITY REVIEW FOR

| | |
|---|---|
| **Prepared for:** | Hubble Exchange |
| **Prepared by:** | Sherlock |
| **Lead Security Expert:** | 0x52 |
| **Dates Audited:** | June 21 - July 3, 2023 |
| **Prepared on:** | August 14, 2023 |

# Introduction

Hubble Exchange is the first Multi-collateral / Cross-Margin Perpetual Futures protocol on Avalanche.

## Scope

Repository: hubble-exchange/hubble-protocol

Branch: main

Commit: d89714101dd3494b132a3e3f9fed9aca4e19aef6

---

For the detailed scope, see the contest details.

## 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.

## Issues found

| Medium | High |
|---|---|
| 13 | 3 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

n1punp
moneyversed
p-tsanev
0xbepresent
PRAISE
yy
shtesesamoubiq
tsvetanovv
ver0759
BugHunter101
MohammedRizwan
dirk_y

Breeje
Vagner
oakcobalt
Bauer
lil.eth
qckhp
p12473
kutugu
rogue-lion-0619
0xDjango
Hama
dimulski
Kaiziron
0xmuxyz
0x52
crimson-rat-reach
osmanozdemir1
Delvir0
0x3e84fa45
ni8mare
qbs
darkart
carrotsmuggler
0xvj
0xpinky
BugBusters
minhtrng
Bauchibred
lemonmon

# Issue H-1: ProcessWithdrawals is still DOS-able

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/116

## Found by

0x3e84fa45, 0x52, 0xDjango, 0xbepresent, carrotsmuggler, dirk_y, kutugu, p12473, qbs, qckhp, rogue-lion-0619

## Summary

DOS on process withdrawals were reported in the previous code4rena audit however the fix does not actually stop DOS, it only makes it more expensive. There is a much cheaper way to DOS the withdrawal queue - that is by specifying the `usr` to be a smart contract that consumes all the gas.

## Vulnerability Detail

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.9;
import "./Utils.sol";

contract MaliciousReceiver {
    uint256 public gas;
    receive() payable external {
        gas = gasleft();
        for(uint256 i = 0; i < 150000; i++) {} // 140k iteration uses about 28m
↪   gas. 150k uses slightly over 30m.
    }
}

contract VUSDWithReceiveTest is Utils {
    event WithdrawalFailed(address indexed trader, uint amount, bytes data);

    function setUp() public {
        setupContracts();
    }

    function test_CannotProcessWithdrawals(uint128 amount) public {
        MaliciousReceiver r = new MaliciousReceiver();

        vm.assume(amount >= 5e6);
        // mint vusd for this contract
        mintVusd(address(this), amount);
        // alice and bob also mint vusd
```

```
        mintVusd(alice, amount);
        mintVusd(bob, amount);

        // withdraw husd
        husd.withdraw(amount);      // first withdraw in the array
        vm.prank(alice);
        husd.withdraw(amount);
        vm.prank(bob);              // Bob is the malicious user and he wants to
↪   withdraw the VUSD to his smart contract
        husd.withdrawTo(address(r), amount);

        assertEq(husd.withdrawalQLength(), 3);
        assertEq(husd.start(), 0);

        husd.processWithdrawals();  // This doesn't fail on foundry because
↪   foundry's gas limit is way higher than ethereum's.

        uint256 ethereumSoftGasLimit = 30_000_000;
        assertGt(r.gas(), ethereumSoftGasLimit); // You can only transfer at
↪   most 63/64 gas to an external call and the fact that the recorded amt of gas
↪   is > 30m shows that processWithdrawals will always revert when called on
↪   mainnet.
    }

    receive() payable external {
        assertEq(msg.sender, address(husd));
    }
}
```

Copy and paste this file into the test/foundry folder and run it.

The test **will not fail** because foundry has a very high gas limit but you can see from the test that the amount of gas that was recorded in the malicious contract is higher than 30m (which is the current gas limit on ethereum). If you ran the test by specifying the —gas-limit i.e. `forge test -vvv --match-path test/foundry/VUSDRevert.t.sol --gas-limit 30000000` The test will fail with `Reason: EvmError: OutOfGas` because there is not enough gas to transfer to the malicious contract to run 150k iterations.

## Impact

Users will lose their funds and have their VUSD burnt forever because nobody is able to process any withdrawals.

## Code Snippet

https://github.com/hubble-exchange/hubble-protocol/blob/d89714101dd3494b132a3e3f9fed9aca4e19aef6/contracts/VUSD.sol#L65-L85

## Tool used

Manual Review

## Recommendation

From best recommendation to worst

1. Remove the queue and withdraw the assets immediately when `withdraw` is called.
2. Allow users to process withdrawals by specifying the index index
3. Allow the admin to remove these bad withdrawals from the queue
4. Allow the admin to adjust the start position to skip these bad withdrawals.

## Discussion

**P12473**

Escalate

I think we should differentiate between a temporary DOS like #153 vs a permanent DOS like my issue (and a few others that are marked as dups of #153 e.g. #57) and the fix suggested by #153 does not fix our vulnerability.

**sherlock-admin2**

> Escalate
>
> I think we should differentiate between a temporary DOS like #153 vs a permanent DOS like my issue (and a few others that are marked as dups of #153 e.g. #57) and the fix suggested by #153 does not fix our vulnerability.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

**ctf-sec**

Please check:

https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/153#issuecomment-1643620806

**P12473**

> Please check:
>
> #153 (comment)

The lack of reentrancy guards does not prevent the attack vector mentioned here. I see 3 broad categorization of issues that can DOS the processWithdraw function.

() Vulnerability Issues Effect

()

| | |
|---|---|
| Consuming more than the max allowed gas limit of the network | #116, #57, #138, #160, #191, #2 |
| Cross functional reentrancy | #153, #195 |
| Spamming the withdrawal queue | everything else |

()

Please note that this issue (and all its dups) will break the `processWithdrawal` function entirely and forever. You cannot undo it because there is no way to process a withdrawal transaction that consumes more gas than is maximally allowed in a single block e.g. 30m on ethereum, 15m on avalanche.

* I have also grouped #158 together because it is effectively the same thing i.e. the miners cannot process the withdrawal because it consumes too much gas (either from loading too much memory or from trying to execute a million iterations of a loop).

**ctf-sec**

This issues just outline an different way to waste gas in external call

Recommend maintaining the original duplication because the issue are grouped by root cause, not by fix or impact

**hrishibhat**

Result: High Has duplicates While I agree given the nature of the issues around external `.call` and out-of-gas-related issues in the `processWithdrawals` function, there could have been multiple versions of duplications. Tend to agree with the analysis made in this comment here. After looking at all the issues and considering the attack vectors mentioned, based on the Sherlock rules this is the version of duplication that was concluded:

- 1. Issues that result in a user being a malicious contract that consumes all gas during `processWithdrawals` #116, #57, #138, #160, #191, #215, #158, #122, #96, #95, #127, #198
- 2. Griefing due to cross-functional reentrancy #153, #195
- 3. DOS due to large withdrawal requests #81, #94, #15

Note: I agree that there may be other duplication sets based on the interpretations of the duplication rules, but this is the one I decided to go with and consider fair.

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:

- p12473: accepted

**asquare08**

Fixed in this PR. The description is in the PR.

**IAm0x52**

Fix looks good. The `call` in `processWithdrawals` now limits gas consumption to `maxGas` which can be adjusted by governance. A optional change I would recommend is to prevent `maxGas` from being set to less than 3000 in the event that admin is compromised in some way.

**asquare08**

Yes. Good suggestion. Will add this with upcoming fixes.

# Issue H-2: Failed withdrawals from VUSD#processWithdrawals will be lost forever

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/162

## Found by

0x52, 0xDjango, 0xpinky, Delvir0, Kaiziron, dirk_y, kutugu, n1punp, ver0759, yy

## Summary

When withdrawals fail inside VUSD#processWithdrawals they are permanently passed over and cannot be retried. The result is that any failed withdrawal will be lost forever.

## Vulnerability Detail

VUSD.sol#L75-L81

```
(bool success, bytes memory data) = withdrawal.usr.call{value:
↪  withdrawal.amount}("");
if (success) {
    reserve -= withdrawal.amount;
} else {
    emit WithdrawalFailed(withdrawal.usr, withdrawal.amount, data);
}
i += 1;
```

If the call to withdrawal.usr fails the contract will simply emit an event and continue on with its cycle. Since there is no way to retry withdrawals, these funds will be permanently lost.

## Impact

Withdrawals that fail will be permanently locked

## Code Snippet

VUSD.sol#L65-L85

## Tool used

Manual Review

## Recommendation

Cache failed withdrawals and allow them to be retried or simply send VUSD to the user if it fails.

## Discussion

**asquare08**

will add functionality to retry withdrawals

**asquare08**

Fixed in this PR. The description is in the PR.

**IAm0x52**

Fix looks good. Failed withdrawals are cached allowing them to be retried by governance on a case by case basis.

# Issue H-3: Rogue validators can manipulate funding rates and profit unfairly from liquidations

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/183

## Found by

0x52

## Summary

Validators are given the exclusive privilege to match. Any validator can abuse this privilege to manipulate funding rates and profit unfairly from liquidations. Normally validators are fully trusted but in the circumstances of these smart contracts and the chain it's deployed on, practically anyone can become a validator and abuse this privilege.

## Vulnerability Detail

Consider the following attack vectors:

1) Profiting unfairly from liquidation. Assume the current mark price for WETH is 2000. A user is liquidated and 100 ETH needs to be sold. Assume there are available orders that can match this liquidation at 2000. Instead of matching it directly, the validator can create his own order at 1980 and match the liquidation to his own order. He can then immediately sell and match with the other available orders for a profit of 2,000 (20 * 100) USDC.
2) Manipulation of funding rate. Validators are allowed to match non-liquidation orders at up to 20% spread from the oracle. This makes it incredibly easy for validators to manipulate the markPriceTwap. By creating and matching tiny orders at the extremes of this spread they can dramatically skew the funding rate it whatever way they please. Max funding rates can liquidate leveraged positions very quickly allowing that validator to further profit from their liquidations.

Now we can discuss how just about anyone can become a validator and abuse this privilege with zero consequences.

First we need to consider the typical methodology for ensuring validators behave and why NONE of those factors apply in this scenario. 1) "Slash validators that misbehave." Hubble mainnet is a fork of the AVAX C-Chain which is different from most chains in the fact that **AVAX validators can't be slashed**. 2) "Validators are forced to hold a volatile protocol token that would depreciate if they are publicly observed misbehaving." On Hubble mainnet the gas token is USDC so it would not

depreciate in the event that validators misbehave. 3) "Blocks that break consensus rules are rejected." The Hubble exchange smart contracts are not part of the consensus layer so abusing validator privilege as described aren't breaking any consensus rules and would therefore be accepted by honest validators.

Second we consider the ease of becoming a validator. Hubble mainnet is a fork of AVAX and uses it's same consensus mechanism. This allows any node who posts the required stake to become a validator for the network. This allows anyone with access to any decent level of capital to become a validator and begin exploiting users.

## Impact

Validators can manipulate funding rates and profit unfairly from liquidations

## Code Snippet

OrderBook.sol#L215-L258

## Tool used

Manual Review

## Recommendation

The methodology of order matching needs to be rethought to force validators to match fairly and efficiently

## Discussion

**asquare08**

We will fix this with post-mainnet releases. Initially, we are launching with a trusted, closed set of validators and will fix this before we open for public validators. Remarks about point 2. Manipulation of funding rate - for this to happen, a validator will need to place and execute orders for a fairly large amount of time if there are other trades in the system. So this scenario can happen in case of low liquidity in the system.

**P12473**

Escalate

This should not be a high because validators in hubble are decided by the governance which is different from the validators who secure the hubble network.

For this exploit to occur, an existing validator needs to be compromised or a rogue validator added by manipulating the governance vote. Moreover, this attack

assumes that there is only 1 validator when there can be multiple non rogue validators matching orders as well.

**sherlock-admin2**

> Escalate
>
> This should not be a high because validators in hubble are decided by the governance which is different from the validators who secure the hubble network.
>
> For this exploit to occur, an existing validator needs to be compromised or a rogue validator added by manipulating the governance vote. Moreover, this attack assumes that there is only 1 validator when there can be multiple non rogue validators matching orders as well.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

**djb15**

I agree with this escalation. Regarding the manipulation of the funding rate see https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/100#issuecomment-1640389572 which is acknowledged as a low severity issue because you just need 1 non-malicious validator. I believe the same is true for this report where one non malicious validator would prevent the funding rate being manipulated over a significant period of time (in normal market conditions).

EDIT: See https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/87#issuecomment-1644045278 for some related justification

**ctf-sec**

In the part funding manipulation yes

but the attacker vector does not require all validator to be malicious, one malicious validator can do this and extract value from user.

> Profiting unfairly from liquidation. Assume the current mark price for WETH is 2000. A user is liquidated and 100 ETH needs to be sold. Assume there are available orders that can match this liquidation at 2000. Instead of matching it directly, the validator can create his own order at 1980 and match the liquidation to his own order. He can then immediately sell and match with the other available orders for a profit of 2,000 (20 * 100) USDC.

Recommend maintaining high severity

**djb15**

That's a fair point, I've found the following quote from the sponsor in the discord channel (https://discord.com/channels/812037309376495636/1121092175216787507/1126818439646957628):

> So, in that sense and from the perspective of the smart contract audit, validators are not trusted.

So I agree the "profit unfairly from liquidations" part of this report is probably a valid high as I agree you only need 1 malicious validator. But the "manipulating funding rates part" is probably a medium as you would need either lots of malicious validators in normal market conditions or 1 malicious validator in low liquidity market conditions. So high overall :D

**IAm0x52**

The validators aren't different, they are the same. The concept that governance would restrict validators to a trusted set is new information that was never present in any of the docs or readme during the contest. All information indicated that validators would be open and decentralized. Based on that information anyone could be a validator which is why I think this should remain as a high risk issue.

#87 should not be a dupe of this issue. Realistically I shouldn't have submitted this as a single issue and should have submitted it as two. One for liquidation manipulation and the other for the funding rate manipulation. Since a single issue can't be a dupe of two different issues, this should remain a separate issue.

**P12473**

> The validators aren't different, they are the same.

If they are the same and any validators of the network can participate in matching orders, then why was this information not communicated to us on the docs or the README?

Can you please show me an example of a smart contract on AVAX C-Chain with a protected "validator" role in the smart contract and yet any of the validators of AVAX consensus layer can pass that check.

More generally, should vulnerabilities regarding information that was not communicated to us be also considered valid? This encompasses literally everything, and you start to go into the world of crazy hypothetical what-ifs. @hrishibhat

> All information indicated that validators would be open and decentralized.

The README clearly states that there is a governance role which is a TRUSTED role. All actions of a trusted role should be trusted hence one would assume that who the governance selects as validators are to be trusted.

If this premise is wrong, then one can go so far as to generalise that all privileged

roles cannot be trusted because there is a non-zero chance that they will go rogue / fall in the hands of the wrong people.

> Realistically I shouldn't have submitted this as a single issue and should have submitted it as two. One for liquidation manipulation and the other for the funding rate manipulation.

Both liquidation manipulation and funding rate manipulation is predicated on the assumption that rogue validators
were selected to match orders by the governance.

Liquidation manipulation also assumes that:

1. there are no other honest validators to match this user.
2. the rogue validator of the smart contract also controls a substantial AVAX stake as a node at the consensus layer in order to ensure that his orders are finalized.

As described by the sponsor, funding rate manipulation can only occur when there is low liquidity. When there is high liquidity, manipulating the TWAP price will be too expensive just to extract funding from users. In many past contests, a comment like this by the sponsor would immediately invalidate the report.

It is also by design that if the system has low liquidity, the funding paid by the user will be high. The onus should be on the user to close his position.

> Since a single issue can't be a dupe of two different issues, this should remain a separate issue.

Lol I'm gonna start grouping all my issues together too because multiple issues cant be a dup of a single issue.

**IAm0x52**

> If they are the same and any validators of the network can participate in matching orders, then why was this information not communicated to us on the docs or the README

All network validators have to be added by governance or else there would be a lot of empty blocks because the unverified block producers wouldn't be able to match any orders. Fundamentally they have to be the same.

> there are no other honest validators to match this user.

This statement is incorrect. When it is this validator's turn to produce the block they will be able to abuse this.

> the rogue validator of the smart contract also controls a substantial AVAX stake as a node at the consensus layer in order to ensure that his orders are finalized.

Also incorrect. If you would read my issue, abusing liquidations in this way doesn't break any consensus rules and therefore would be accepted by honest validators.

> Both liquidation manipulation and funding rate manipulation is predicated on the assumption that rogue validators
> were selected to match orders by the governance.

Incorrect again. Anyone, not just validators, can manipulate funding rate as demonstrated by #87. Hence why this is two separate issues.

> Lol I'm gonna start grouping all my issues together too because multiple issues cant be a dup of a single issue.

Yet another incorrect statement. In this scenario I receive less of the prize pool not more.

**hrishibhat**

@P12473 I think the above response makes sense. do you have further comments?

**P12473**

> @P12473 I think the above response makes sense. do you have further comments?

Thanks for giving me this opportunity to speak but I do not have any comments.

@IAm0x52 is a much better auditor than I am and it is most likely that I am wrong. I want to understand his perspective but I am unable to do so from his explanation. If you agree with his take then I will rest my case.

I only hope that the decision was an impartial one. Thanks again!

**asquare08**

For the scope of this audit, validators should not be trusted. However, we'll launch on mainnet with a closed set of trusted validators and plan to open for public validators later. So, point 1 (liquidation) is a valid high issue as it can happen even if 1 validator is malicious and point 2 (funding payment) is a valid medium issue as it can happen only in low liquidity case and will also be costly to do so.

**hrishibhat**

Result: High Unique Considering this issue a valid high based on the discussion above and Sponsor's comment

Additionally, the readme shared information about the trusted protocol roles were limited to the governance. And the docs provided context on the validators and their functioning: https://medium.com/hubbleexchange/introducing-hubble-exchange-a-purpose-built-chain-for-perps-25c60db66d44 https://hubbleexchange.notion.site/Hubble-Exchange-Overview-e9487ec19fe9451c9d445be15978c740

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:

- p12473: rejected

**asquare08**

As mentioned in the above comment, we will fix this in later releases

# Issue M-1: Risk of Unfair Order Execution Price in `_validateOrdersA` Function

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/8

## Found by

moneyversed

## Summary

While the system seems to have appropriate checks in place for order matching, a potential risk exists around the order execution pricing logic, specifically when the orders are placed in the same block. In such cases, the system currently selects the sell order price as the fill price. This may not be fair for the buyer, who may end up paying more than necessary.

## Vulnerability Detail

In the `_validateOrdersAndDetermineFillPrice` function, when two orders are placed in the same block, the fill price is set as the price of the sell order (orders[1]). This is potentially unfair because it could lead to a scenario where the buyer is paying the maximum possible price, even when the sell order was potentially willing to sell at a lower price.

## Impact

This could affect the trust of participants in the exchange, particularly those who frequently make buy orders. The unfair pricing might result in monetary losses for these participants, leading to a negative perception of the platform.

## Code Snippet

```
if (blockPlaced0 < blockPlaced1) {
    mode0 = OrderExecutionMode.Maker;
    fillPrice = orders[0].price;
} else if (blockPlaced0 > blockPlaced1) {
    mode1 = OrderExecutionMode.Maker;
    fillPrice = orders[1].price;
} else { // both orders are placed in the same block, not possible to determine
↪   what came first in solidity
    // executing both orders as taker order
    mode0 = OrderExecutionMode.SameBlock;
```

```
    mode1 = OrderExecutionMode.SameBlock;
    // Bulls (Longs) are our friends. We give them a favorable price in this
↪   corner case
    fillPrice = orders[1].price;
}
```

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/orderbooks/OrderBook.sol#L288-L300

## Tool used

Manual Review

## Recommendation

A more fair approach might be to calculate the average of the buy order and sell order prices, which might better reflect the market conditions at the moment both orders were made. In case of any restrictions for using the average price, a price setting rule that better approximates a fair market price should be used.

## Proof Of Concept

To reproduce this vulnerability:

1. Deploy the smart contract on a local testnet or Ethereum mainnet fork.
2. Place two orders: a buy order and a sell order within the same block with different prices.
3. Once the orders are matched and executed, check the execution price.
4. You will notice that the execution price is exactly the same as the sell order, regardless of the buy order's price.

Note: This is a potential risk, meaning that it depends on the intentions of the traders when they place their orders. Traders who are willing to trade at the execution price would not consider this a problem, while others might see it as an unfair pricing practice.

## Discussion

**asquare08**

In the case of same block trade, shorts are selling at the price they quoted and longs are buying at price  long order price. As it is a limit order system, the user get to decide the price of the order and they are always getting a better or the same price than what they quoted. There is no scenario where a user is getting unfair/worst price.

**ctf-sec**

Although the issue has sponsor dispute tag, I am more incline with the watson and leave this as a medium issue

Because

> When a validator is selected as the block producer, the buildBlock function fetches active markets and open orders from the indexer, evaluates open positions for potential liquidations, runs the matching engine, and then relays these operations as local transactions before continuing the normal transaction bundling process.

the block producer may determine the order matching and manipulate who is order[0] and who is order[1] to extract value

```
{ // both orders are placed in the same block, not possible to determine what
↪  came first in solidity
            // executing both orders as taker order
            mode0 = OrderExecutionMode.SameBlock;
            mode1 = OrderExecutionMode.SameBlock;
            // Bulls (Longs) are our friends. We give them a favorable price in
↪  this corner case
            fillPrice = orders[1].price;
        }
```

Want to quote the first second in the report:

https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/183

> Validators are given the exclusive privilege to match.

**asquare08**

Yes correct validator can extract MEV from this by opening longs at better price everytime. Changed status to confirmed. FYI: Fill price determination happens through precompile now and long order is executed as taker and short as maker (maker fee is less than taker fee). So long will get better price but pay more fee and short order will execute at order price but pay less fee

**asquare08**

We'll be launching with a trusted set of validators. So, we'll fix this in later releases when we allow anyone to become a validator.

# Issue M-2: `Chainlink.latestRoundData()` may return stale results

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/18

## Found by

0x3e84fa45, 0xbepresent, 0xmuxyz, 0xpinky, 0xvj, Bauer, Breeje, BugBusters, BugHunter101, Hama, Kaiziron, MohammedRizwan, PRAISE, Vagner, carrotsmuggler, crimson-rat-reach, darkart, dimulski, dirk_y, kutugu, lemonmon, lil.eth, minhtrng, osmanozdemir1, p-tsanev, rogue-lion-0619, shtesesamoubiq, tsvetanovv

## Summary

The Oracle.getUnderlyingPrice() function is used to get the price of tokens, the problem is that the function does not check for stale results.

## Vulnerability Detail

The Oracle.getUnderlyingPrice() function is used in InsuranceFund, MarginAccount and AMM contracts. The `Oracle.getUnderlyingPrice()` helps to determine the tokens prices managed in the contracts.

The problem is that there is not check for stale data. There are some reasons that the price feed can become stale.

## Impact

Since the token prices are used in many contracts, stale data could be catastrophic for the project.

## Code Snippet

The Oracle.getUnderlyingPrice() is used by the next contracts:

- InsuranceFund
- MarginAccount
- AMM

## Tool used

Manual review

## Recommendation

Read the updatedAt return value from the `Chainlink.latestRoundData()` function and verify that is not older than than specific time tolerance.

```
require(block.timestamp - udpatedData < toleranceTime, "stale price");
```

## Discussion

**asquare08**

will add a tolerance time equal to the max update time of the price feed.

**asquare08**

Fixed in this PR. The description is in the PR.

**IAm0x52**

Fix looks good. `chainLinkAggregatorMap` now stores a struct which contains the oracle address and the heartbeat. `getUnderlyingPrice` will now revert if price update is outside of heartbeat window

# Issue M-3: Setting stable price in oracle can result in arbitrate opportunities and significant bad debt if the stable depegs

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/69

## Found by

Bauchibred, dirk_y, kutugu, lil.eth, minhtrng, n1punp, oakcobalt, p12473, rogue-lion-0619

## Summary

In `Oracle.sol` there is the ability for governance to manually specify the price of an underlying asset (designed for stable coins). However, when the given stable coin depegs there is now the opportunity for arbitrage and even the possibility that Hubble becomes insolvent.

## Vulnerability Detail

Firstly, it is worth noting that stable coins depeg relatively frequently. For the sake of argument let's say that a stable coin is pegged to the US Dollar and the price is set by governance to `1000000` (to 6 decimal places) in the oracle.

Probably the most important place in the protocol where the price of an asset is used is in `MarginAccount` in the `weightedAndSpotCollateral` method. This method is used under the hood to check whether a user can withdraw from the margin account and whether the user is able to be liquidated. It is also used in the AMM logic but I'll focus on the margin account case as I think that is the easiest to exploit.

Let's now say that the stable coin depegs from its $1 price. Now, a user that has already used this stable coin as collateral in the margin account should have a lower value of collateral and therefore could be at risk of being liquidated. However, because the price of the stable coin is manually pegged to $1, the unhealthy trader's position will still appear healthy. If the stable coin failed to return to it's previous $1 value then even if the price of the stable coin was changed by governance to reflect the new lower value, the amount of bad debt accrued by Hubble would be huge at the time of liquidation/settlement.

The other scenario that will be used more actively by malicious users is that they will acquire the depegged stable coin from another source at its depegged value (e.g. $0.9). The malicious user can then deposit this stable coin into the Hubble margin account at the hard coded value of $1, and open positions with a value that should ordinarily put the trader into a bad debt position.

## Impact

A short term stable coin depeg event will result in accounts not being liquidated when they should be, and it will allow users to gain from arbitrage trades where they purchase a stable coin at its depegged price and are offered a higher price in Hubble.

A permanent depeg will result in a huge amount of bad debt in Hubble and would likely cause the protocol to become insolvent due to the inability to perform liquidations during the downward price movement.

## Code Snippet

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/Oracle.sol#L30-L32 https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/Oracle.sol#L172

## Tool used

Manual Review

## Recommendation

I would recommend not having the option to manually set and read the price of a stable coin. Yes, you protect accounts from being liquidated from a short term depeg event, however it is precisely in these volatile market periods that accounts should be able to be liquidated to prevent Hubble from accruing too much bad debt, particularly if the depeg is permanent, in which case it will likely cause Hubble to become insolvent.

## Discussion

**asquare08**

We are using stable price mechanism just for our testnet. We will use actual USDC price for the mainnet.

**asquare08**

As mentioned in the above comment, there is no change required for this in contracts. We'll set stable price for usdc to 0 to use oracle price feeds for usdc.

**IAm0x52**

No fix needed in the current contract. Mainnet deployment will use oracle feed for USDC

**MLON33**

No fix will be made so Sherlock considers this issue to be acknowledged by the protocol team.



# Issue M-4: Malicious user can frontrun withdrawals from Insurance Fund to significantly decrease value of shares

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/72

## Found by

0xDjango, Delvir0, carrotsmuggler, dirk_y, lil.eth, rogue-lion-0619

## Summary

When a user withdraws from the insurance fund, the value of their shares is calculated based on the balance of vUSD in the fund. Another user could deliberately frontrun (or frontrun by chance) the withdrawal with a call to `settleBadDebt` to significantly reduce the vUSD returned from the withdrawal with the same number of shares.

## Vulnerability Detail

When a user wants to withdraw from the insurance pool they have to go through a 2 step withdrawal process. First they need to unbond their shares, and then they have to wait for the pre-determined unbonding period before they can withdraw the vUSD their shares are worth by calling `withdraw`.

When a user calls `withdraw` the amount of vUSD to redeem is calculated as:

```
amount = balance() * shares / totalSupply();
```

where `balance()` is the balance of vUSD in the contract and `totalSupply()` is the total supply of share tokens. Therefore, if the balance of vUSD in the contract were to decrease, then the amount of vUSD redeemed from the same number of shares would decrease as a result.

This occurs when a trader's bad debt is settled when calling `settleBadDebt` in `MarginAccount.sol` as this calls `insuranceFund.seizeBadDebt` under the hood, which in turn calls `settlePendingObligation` which transfers vUSD out of the insurance fund to the margin account:

```
vusd.safeTransfer(marginAccount, toTransfer);
```

The result is now that the balance of vUSD in the insurance fund is lower and thus the shares are worth less vUSD as a consequence.

## Impact

A user withdrawing from the insurance fund could receive significantly less (potentially 0) vUSD when finalising their withdrawal.

## Code Snippet

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/InsuranceFund.sol#L215 https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/InsuranceFund.sol#L259-L261 https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/InsuranceFund.sol#L153-L161

## Tool used

Manual Review

## Recommendation

One option would be to include a slippage parameter on the `withdraw` and `withdrawFor` methods so that the user redeeming shares can specify the minimum amount of vUSD they would accept for their shares.

When depositing into the insurance fund, the number of shares to mint is actually calculated based on the total value of the pool (value of vUSD and all other collateral assets). Therefore, the withdraw logic could also use `_totalPoolValue` instead of `balance()` to get a "true" value per share, however this could lead to withdrawals failing while assets are up for auction. Assuming all the assets are expected to be sold within the short 2 hour auction duration, this is probably the better solution given the pricing is more accurate, but it depends if users would accept failed withdrawals for short periods of time.

## Discussion

**asquare08**

Withdrawals done after `settleBadDebt` and before the end of the auction of the seized collateral, will receive less USDC as compared to withdrawals done after the auction has ended. However, a user deliberately cannot front run all withdrawals with `settleBadDebt` as it will required generating bad debt first. Hence, the severity can be changed to `low`

**ctf-sec**

Emm worth checking the duplicate as well, the duplicate highlights the total supply can be inflated to make user mint less share as well

root cause is the totalSupply() can be inflated by transferring asset directly

for example duplicate #144, #224

recommend maintaining high severity

**ctf-sec**

Comment from senior Watson

Total supply can be inflated by donation but that would distribute the funds to other users in the pool making this a bigger loss than any kind of gain.

Assuming they owned the entire pool, which the only way to make donating profitable, the victim wouldn't lose any value because the donated assets would be auctioned.

When burning, donation of other assets doesn't affect the amount of vUSD received.

Best case I see for this is medium since there is potential for withdraws to be frontrun by calls to settle bad debt. Since there is no way to create bad debt arbitrarily it would be dependent on bad debt already existing but being unclaimed:

**asquare08**

agree with the above commnet

**asquare08**

As this case arises only in case of multi-collateral, we will fix this in later releases

# Issue M-5: min withdraw of 5 VUSD is not enough to prevent DOS via VUSD.sol#withdraw(amount)

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/81

## Found by

0xbepresent, lil.eth, p-tsanev

## Summary

A vulnerability exists where a malicious user spam the contract with numerous withdrawal requests (e.g., 5,000). This would mean that genuine users who wish to withdraw their funds may find themselves unable to do so in a timely manner because the processing of their withdrawals could be delayed significantly.

## Vulnerability Detail

The issue stems from the fact that there is no restriction on the number of withdrawal requests a single address can make. A malicious actor could repeatedly call the withdraw or withdrawTo function, each time with a small amount (min 5 VUSD), to clog the queue with their withdrawal requests.

```
//E Burn vusd from msg.sender and queue the withdrawal to "to" address
function _withdrawTo(address to, uint amount) internal {
    //E check min amount
    require(amount >= 5 * (10 ** 6), "min withdraw is 5 vusd"); //E @audit-info
↪   not enough to prevent grief
    //E burn this amount from msg.sender
    burn(amount); // burn vusd from msg.sender
    //E push
    withdrawals.push(Withdrawal(to, amount * 1e12));
}
```

Given the maxWithdrawalProcesses is set to 100, and the withdrawal processing function processWithdrawals doesn't have any parameter to process from a specific index in the queue, only the first 100 requests in the queue would be processed at a time.

```
uint public maxWithdrawalProcesses = 100;
//E create array of future withdrawal that will be executed to return
function withdrawalQueue() external view returns(Withdrawal[] memory queue) {
    //E check if more than 100 requests in withdrawals array
    uint l = _min(withdrawals.length-start, maxWithdrawalProcesses);
```

```
        queue = new Withdrawal[](l);

        for (uint i = 0; i < l; i++) {
            queue[i] = withdrawals[start+i];
        }
    }
```

In the case of an attack, the first 100 withdrawal requests could be those of the attacker, meaning that the genuine users' requests would be stuck in the queue until all of the attacker's requests have been processed. Moreover the fact that we can only withdraw up to 1 day long when our withdraw request is good to go.

## Impact

This could result in significant delays for genuine users wanting to withdraw their funds, undermining the contract's usability and users' trust in the platform.

## Code Snippet

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/VUSD.sol#L88

## Tool used

Manual Review

## Recommendation

Either limit number of withdrawal requests per address could be a first layer of defense even if it's not enough but I don't see the point why this limit is included so removing it could mitigate this. Otherwise you could implement a priority queue regarding amount to be withdrawn

## Discussion

**asquare08**

Since this is a temporary DOS, we will increase `maxWithdrawalProcesses` if such a condition arises or increase the min withdrawal amount to increase the capital requirement to do so.

# Issue M-6: Malicious user can control premium emissions to steal margin from other traders

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/87

## Found by

dirk_y

## Summary

A malicious user can force premiums to be applied in a positive direction for their positions. They can effectively steal margin from other traders that have filled the other side of their positions.

## Vulnerability Detail

This vulnerability stems from how the premiums are calculated when `settleFunding` is called in `AMM.sol`:

```
int256 premium = getMarkPriceTwap() - underlyingPrice;
```

Effectively, the premium for a position is calculated based on the difference between the perpetual maker TWAP and the oracle TWAP. Under the hood, `getMarkPriceTwap` calls `_calcTwap`, which calculates the TWAP price from the last hour to the current block timestamp:

```
uint256 currentPeriodStart = (_blockTimestamp() / spotPriceTwapInterval) *
↪  spotPriceTwapInterval;
uint256 lastPeriodStart = currentPeriodStart - spotPriceTwapInterval;

// If there is no trade in the last period, return the last trade price
if (markPriceTwapData.lastTimestamp <= lastPeriodStart) {
    return markPriceTwapData.lastPrice;
}

/**
* check if there is any trade after currentPeriodStart
* since this function will not be called before the nextFundingTime,
* we can use the lastPeriodAccumulator to calculate the twap if there is a trade
↪  after currentPeriodStart
*/
if (markPriceTwapData.lastTimestamp >= currentPeriodStart) {
    // use the lastPeriodAccumulator to calculate the twap
    twap = markPriceTwapData.lastPeriodAccumulator / spotPriceTwapInterval;
```

```
} else {
    // use the accumulator to calculate the twap
    uint256 currentAccumulator = markPriceTwapData.accumulator +
↪   (currentPeriodStart - markPriceTwapData.lastTimestamp) *
↪   markPriceTwapData.lastPrice;
    twap = currentAccumulator / spotPriceTwapInterval;
}
```

This method works closely in conjunction with `_updateTWAP` which is called every time a new position is opened based on the fill price. I'll talk more about his in the "Recommendation" section, but the core issue is that too much weight is placed on the last price that was filled, along with the fact the user can open uncapped positions. As can be seen from the `_calcTwap` method above, if there has not been a recently opened position, then the TWAP is determined as the last filled price. And naturally, a time weighted price isn't weighted by the size of a fill as well, so the size of the last fill has no impact.

As a result of this, a malicious user can place orders (which should then be executed by the validators) at a price that maximises the difference between the market TWAP and the oracle TWAP in order to maximise the premiums generated in the market. If the malicious user opens up a large enough position, the premiums generated exceed the taker/maker fees for opening positions. And since the same user can place orders for both sides of the market, they do not need to increase their margin requirement over time in order to meet the minimum margin requirements. Effectively the user is able to generate free revenue assuming the price of the underlying asset doesn't significantly deviate in the opposite direction of the large position held by the user.

Below is a diff to the existing test suite with a test case that shows how a malicious user could control premiums to make a profit. It can be run with `forge test -vvv --match-path test/foundry/OrderBook.t.sol`:

```
diff --git a/hubble-protocol/test/foundry/OrderBook.t.sol
↪   b/hubble-protocol/test/foundry/OrderBook.t.sol
index b4dafdf..f5d36b2 100644
--- a/hubble-protocol/test/foundry/OrderBook.t.sol
+++ b/hubble-protocol/test/foundry/OrderBook.t.sol
@@ -228,6 +228,60 @@ contract OrderBookTests is Utils {
         assertPositions(bob, -size, quote, 0, quote * 1e18 / stdMath.abs(size));
     }

+    function testUserCanControlEmissions() public {
+        uint256 price = 1e6;
+        oracle.setUnderlyingPrice(address(wavax), int(uint(price)));
+
+        // Calculate how much margin required for 100x MIN_SIZE
```

```
+        uint256 marginRequired = orderBook.getRequiredMargin(100 * MIN_SIZE,
↪  price) * 1e18 / uint(defaultWethPrice) + 1e10; // required weth margin in
↪  1e18, add 1e10 for any precision loss
+
+        // Let's say Alice is our malicious user, and Bob is a normal user
+        addMargin(alice, marginRequired, 1, address(weth));
+        addMargin(bob, marginRequired, 1, address(weth));
+
+        // Alice places a large legitimate long order that is matched with a
↪  short order from Bob
+        placeAndExecuteOrder(0, aliceKey, bobKey, MIN_SIZE * 90, price, true,
↪  false, MIN_SIZE * 90, false);
+
+        // Alice's free margin is now pretty low
+        int256 availabeMargin = marginAccount.getAvailableMargin(alice);
+        assertApproxEqRel(availabeMargin, 200410, 0.1e18); // Assert within 10%
+
+        // Calculate what's the least we could fill an order for given the
↪  oracle price
+        uint256 spreadLimit = amm.maxOracleSpreadRatio();
+        uint minPrice = price * (1e6 - spreadLimit) / 1e6;
+
+        // Alice can fill both sides of an order at the minimum fill price
↪  calculated above, with the minimum size
+        // Alice would place such orders (and hopefully have them executed)
↪  just after anyone else makes an order in a period (1 hour)
+        // The goal for Alice is to keep the perpetual TWAP as low as possible
↪  vs the oracle TWAP (since she holds a large long position)
+        // In quiet market conditions Alice just has to make sure she's the
↪  last person to fill
+        // In busy market conditions Alice would fill an order immediately
↪  after anyone else fills an order
+        // In this test Alice fills an order every 2 periods, but in reality,
↪  if nobody was trading then Alice wouldn't have to do anything provided she
↪  was the last filler
+        for (uint i = 0; i < 100; i++) {
+            uint256 currentPeriodStart = (block.timestamp / 1 hours) * 1 hours;
+
+            // Warp to before the end of the period
+            vm.warp(currentPeriodStart + 3590);
+
+            // Place and execute both sides of an order as Alice
+            // Alice can do this because once both sides of the order are
↪  executed, the effect to her free margin is 0
+            // As mentioned above, Alice would place such orders every time
↪  after another order is executed
+            placeAndExecuteOrder(0, aliceKey, aliceKey, MIN_SIZE, minPrice,
↪  true, false, MIN_SIZE, false);
```



```
+
+            // Warp to the start of the next period
+            vm.warp(currentPeriodStart + (3600 * 2) + 10);
+
+            // Funding is settled. This calculates the premium emissions by
 ↪  comparing the perpetual twap with the oracle twap
+            orderBook.settleFunding();
+        }
+
+        // Alice's margin is now significantly higher (after just 200 hours)
 ↪  because she's been pushing the premiums in her direction
+        availabeMargin = marginAccount.getAvailableMargin(alice);
+        assertApproxEqRel(availabeMargin, 716442910, 0.1e18); // Assert within
 ↪  10%
+
+    }
+
     function testLiquidateAndExecuteOrder(uint64 price, uint120 size_) public {
         vm.assume(price > 10 && size_ != 0);
         oracle.setUnderlyingPrice(address(wavax), int(uint(price)));
```

## Impact

A user can effectively steal funds from other traders that are filling the other side of their positions. The larger the position the malicious user is able to fill and the longer the period, the more funds can be credited to the malicious user's margin account.

## Code Snippet

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/AMM.sol#L255-L258 https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/AMM.sol#L501-L503

## Tool used

Manual Review

## Recommendation

I originally thought the best way to mitigate this kind of attack is to scale the TWAP calculation based on the filled amount vs the total fill amount of the whole market. However the downside with this approach is that the fill amount will perpetually increase (given it's a perpetual market after all!) and so the market TWAP

deviations from the oracle TWAP would decrease and so the premium emissions would also decrease over time. This could be argued as a feature in that early users receive a larger premium than later users.

Upon further thought I think the best way to prevent this kind of attack is simply to disincentivise the malicious user from doing so; by making this a net-loss situation. This can be done with a combination of the following:

- Increasing minimum order size
- Increasing trader/maker fees
- Introducing another fixed fee per order (rather than only variable rate fees)
- Capping the maximum position size (both long and short)
- Reducing the maximum price deviation of fill prices from oracle price
- Increasing the minimum margin requirements

This will vary per perpetual market, but the key thing that needs to be accomplished is that the cost to a user to place orders to control the market TWAP is greater than the premium that can be obtained from their position. This will also require some estimates as to how frequently users are going to be placing orders. If orders are relatively infrequent then increasing the TWAP calculation from 1 hour will also help with this.

It is also worth considering whether the following lines in `_calcTwap` are overly weighted towards the last fill price:

```
// If there is no trade in the last period, return the last trade price
 if (markPriceTwapData.lastTimestamp <= lastPeriodStart) {
     return markPriceTwapData.lastPrice;
 }
```

You could make the argument that if no trades have occurred in a significant period of time then the market TWAP should revert back to the oracle TWAP and premium emissions should halt. This could either be after one empty period, or X number of empty periods to be defined by Hubble.

Finally, having a trader able to hold both sides of the same perpetual in the same order makes this attack easier to implement, so it might be worth adding an extra check to prevent this. However it's worth noting the same could be achieved with 2 accounts assuming they alternated the long/short positions between them to avoid excessive margin requirements. So I'm not sure this is strictly necessary.

## Discussion

**asquare08**

This is a scenario of low liquidity where no trade has happened in the last 1 hour and if happened, a malicious user has made a trade just after that to move the price up/down. Many other systems might also fail in such a scenario. Also, only validators can match the placed orders and the malicious user will not always get their desired price unless the validator picks their short and long orders. Hence we can change the severity to `medium`

**ctf-sec**

Changed the severity to medium

**MarkuSchick**

Escalate

The market manipulation the user is referring to is a dublicate of the funding rate manipulation raised in https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/183: `2. Manipulation of funding rate.`

According to other comments funding rate manipulation is a low severity issue.

**sherlock-admin2**

> Escalate
>
> The market manipulation the user is referring to is a dublicate of the funding rate manipulation raised in https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/183: `2. Manipulation of funding rate.`
>
> According to other comments funding rate manipulation is a low severity issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

**djb15**

In my (biased) opinion I think the medium (not low) severity argument for the other issue is that you would need multiple validators to be malicious which is highly unlikely (**in a normal market environment**).

In this issue **any** user can open orders to manipulate the funding rate, but this is only possible in low liquidity scenarios which is why it is a medium severity issue.

A single malicious validator would be able to do the same thing as is detailed in this report in low liquidity market environments, which would make these two issues identical. Hence why I think both should be medium severity.

**asquare08**

As mentioned in the comment here, its a valid medium issue. Also refer to this comment

**hrishibhat**

Result: Medium Unique Considering this a valid medium issue

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:

- MarkuSchick: rejected

# Issue M-7: Malicious user can grief withdrawing users via VUSD reentrancy

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/153

## Found by

0x3e84fa45, 0x52

## Summary

VUSD#processWithdraw makes a call to withdrawal.usr to send the withdrawn gas token. processWithdrawals is the only nonreentrant function allowing a user to create a smart contract that uses it's receive function to deposit then immediately withdraw to indefinitely lengthen the withdrawal queue and waste large amounts of caller gas.

## Vulnerability Detail

VUSD.sol#L69-L77

```
while (i < withdrawals.length && (i - start) < maxWithdrawalProcesses) {
    Withdrawal memory withdrawal = withdrawals[i];
    if (reserve < withdrawal.amount) {
        break;
    }

    (bool success, bytes memory data) = withdrawal.usr.call{value:
    ↪  withdrawal.amount}("");
    if (success) {
        reserve -= withdrawal.amount;
```

To send the withdrawn gas token to the user VUSD#processWithdrawals utilizes a call with no data. When received by a contract this will trigger it's receive function. This can be abused to continually grief users who withdraw with no recurring cost to the attacker. To exploit this the attacker would withdraw VUSD to a malicious contract. This contract would deposit the received gas token then immediately withdraw it. This would lengthen the queue. Since the queue is first-in first-out a user would be forced to process all the malicious withdrawals before being able to process their own. While processing them they would inevitably reset the grief for the next user.

NOTE: I am submitting this as a separate issue apart from my other two similar issues. I believe it should be a separate issue because even though the outcome is

similar the root cause is entirely different. Those are directly related to the incorrect call parameters while the root cause of this issue is that both mintWithReserve and withdraw/withdrawTo lack the reentrant modifier allowing this malicious reentrancy.

## Impact

Malicious user can maliciously reenter VUSD to grief users via unnecessary gas wastage

## Code Snippet

VUSD.sol#L45-L48

VUSD.sol#L50-L52

VUSD.sol#L58-L60

## Tool used

Manual Review

## Recommendation

Add the nonreentrant modifer to mintWithReserve withdraw and withdrawTo

## Discussion

**asquare08**

Will add the `nonreentrant` modifer to `mintWithReserve` withdraw and `withdrawTo`

**ctf-sec**

> To exploit this the attacker would withdraw VUSD to a malicious contract. This contract would deposit the received gas token then immediately withdraw it.

I put this issue and #160 together because these two issue highlight different ways of wasting gas, but they also focus on how to waste gas in external call.

Recommend checking #160 as well.

and I leave the #158 as a separate issue because the root cause is the returned call data is emitted in the contract code itself

**asquare08**

yes noted. #160 has slightly different cause but same effect. So the solution for all these related issues is

- don't save data in variable #158

- cap the gas limit of .call #160
- this

**IAm0x52**

Escalate

As the sponsor has pointed out, this is a different issue from the dupes. While the outcome of wasting gas is similar, the root cause is completely different. The root cause for this is reentrancy across functions, while the root cause of issues marked as dupes is that there is no gas limit. I suggest that this issue be separated and the dupes groped together as separate issues.

Edit: Missed #195. That and this should be considered a separate issue

**sherlock-admin2**

> Escalate
>
> As the sponsor has pointed out, this is a different issue from the dupes. While the outcome of wasting gas is similar, the root cause is completely different. The root cause for this is reentrancy across functions, while the root cause of issues marked as dupes is that there is no gas limit. I suggest that this issue be separated and the dupes groped together as separate issues.
>
> Edit: Missed #195. That and this should be considered a separate issue

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

**ctf-sec**

See escalation
https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/116

There are a lot of ways to consume all gas in external call (reentrancy to expand the queue size, gas token, for loop, swap, etc....), cannot count each of them as duplicates

I think grouping all these issue about wasting gas in external call to one issue make sense, root cause is gas limit not capped.

**0xArcturus**

Agreed with escalation, as also mentioned in #195, the root cause of this is the lack of reentrancy guard and the `withdrawals.length` changing during execution. This attack can bloat the queue by reentering with a single withdraw call, while the dupes are focused on consuming gas in a single call.

**hrishibhat**

Result: Medium Has duplicates Agree with the escalation that this should be a separate issue along with #195

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:

- IAm0x52: accepted

**asquare08**

Fixed in this PR. The description is in the PR.

**IAm0x52**

Fix looks good. `mintWithReserve`, `withdraw`, and `withdrawTo` now use the `nonreentrant` modifier to prevent this exploit

# Issue M-8: Malicious users can donate/leave dust amounts of collateral in contract during auctions to buy other collateral at very low prices

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/168

## Found by

0x3e84fa45, 0x52, rogue-lion-0619, ver0759

## Summary

Auctions are only ended early if the amount of the token being auctioned drops to 0. This can be exploited via donation or leaving dust in the contract to malicious extend the auction and buy further liquidate collateral at heavily discounted prices.

## Vulnerability Detail

InsuranceFund.sol#L184-L199

```
function buyCollateralFromAuction(address token, uint amount) override external {
    Auction memory auction = auctions[token];
    // validate auction
    require(_isAuctionOngoing(auction.startedAt, auction.expiryTime),
    ↪  "IF.no_ongoing_auction");

    // transfer funds
    uint vusdToTransfer = _calcVusdAmountForAuction(auction, token, amount);
    address buyer = _msgSender();
    vusd.safeTransferFrom(buyer, address(this), vusdToTransfer);
    IERC20(token).safeTransfer(buyer, amount); // will revert if there wasn't
    ↪  enough amount as requested

    // close auction if no collateral left
    if (IERC20(token).balanceOf(address(this)) == 0) { <- @audit-issue only
    ↪  cancels auction if balance = 0
        auctions[token].startedAt = 0;
    }
}
```

When buying collateral from an auction, the auction is only closed if the balance of the token is 0. This can be exploited in a few ways to maliciously extend auctions and keep the timer (and price) decreasing. The first would be buy all but 1 wei of a token leaving it in the contract so the auction won't close. Since 1 wei isn't worth

the gas costs to buy, there would be a negative incentive to buy the collateral, likely resulting in no on buying the final amount. A second approach would be to frontrun an buys with a single wei transfer with the same results.

Now that the auction has been extended any additional collateral added during the duration of the auction will start immediately well below the assets actual value. This allows malicious users to buy the asset for much cheaper, causing loss to the insurance fund.

## Impact

Users can maliciously extend auctions and potentially get collateral for very cheap

## Code Snippet

InsuranceFund.sol#L184-L199

## Tool used

Manual Review

## Recommendation

Close the auction if there is less than a certain threshold of a token remaining after it has been bought:

```
    IERC20(token).safeTransfer(buyer, amount); // will revert if there wasn't
    ↪  enough amount as requested

+   uint256 minRemainingBalance = 1 * 10 ** (IERC20(token).decimal() - 3);

    // close auction if no collateral left
+   if (IERC20(token).balanceOf(address(this)) <= minRemainingBalance) {
        auctions[token].startedAt = 0;
    }
```

## Discussion

**asquare08**

This issue can come when multi-collateral is enabled. Therefore, we will fix this with post-mainnet releases as we are launching mainnet with single collateral.

# Issue M-9: MarginAccountHelper will be bricked if registry.marginAccount or insuranceFund ever change

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/170

## Found by

0x52, crimson-rat-reach

## Summary

MarginAccountHelper#syncDeps causes the contract to refresh it's references to both marginAccount and insuranceFund. The issue is that approvals are never made to the new contracts rendering them useless.

## Vulnerability Detail

MarginAccountHelper.sol#L82-L87

```
function syncDeps(address _registry) public onlyGovernance {
    IRegistry registry = IRegistry(_registry);
    vusd = IVUSD(registry.vusd());
    marginAccount = IMarginAccount(registry.marginAccount());
    insuranceFund = IInsuranceFund(registry.insuranceFund());
}
```

When syncDeps is called the marginAccount and insuranceFund references are updated. All transactions require approvals to one of those two contract. Since no new approvals are made, the contract will become bricked and all transactions will revert.

## Impact

Contract will become bricked and all contracts that are integrated or depend on it will also be bricked

## Code Snippet

MarginAccountHelper.sol#L82-L87

## Tool used

Manual Review

## Recommendation

Remove approvals to old contracts before changing and approve new contracts after

## Discussion

**asquare08**

Valid issue but we will be using `syncDeps` mainly during the deployment. Later on, since both `marginAccount` and `insuranceFund` are upgradeable contracts, their address won't change.

# Issue M-10: Funding settlement will be DOS'd for a time after the phaseID change of an underlying chainlink aggregator

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/177

## Found by

0x52

## Summary

Oracle incorrectly assumes that roundID is always incremented by one but this is not the case. Chainlink's roundID is actually two values packed together: phaseId and aggregatorRoundId. When the phaseID is incremented the roundID increases by 2 ** 64. After a phaseID increment all calls to settle funding will revert until an entire funding interval has elapsed. Since all markets are settled simultaneously, even a single oracle incrementing will result in all market funding being DOS'd. Although this funding can be made up later it will user different TWAP values which will result in users being paid differently causing loss of yield to a portion of all users.

## Vulnerability Detail

https://snowtrace.io/address/0x976b3d034e162d8bd72d6b9c989d545b839003b0#code#L206

```
function getAnswer(uint256 _roundId)
  public
  view
  virtual
  override
  returns (int256 answer)
{
  if (_roundId > MAX_ID) return 0;

  (uint16 phaseId, uint64 aggregatorRoundId) = parseIds(_roundId);
  AggregatorV2V3Interface aggregator = phaseAggregators[phaseId];
  if (address(aggregator) == address(0)) return 0;

  return aggregator.getAnswer(aggregatorRoundId);
}
```

The above code is from the ETH/USD aggregator on AVAX, It can be seen that the roundId is made up of 2 packed components, the phaseId and aggregatorRoundId. As explained in the summary, when the phaseId is incremented 2 ** 64 "rounds" will be skipped. When `currentRound - 1` is inevitably queried after this increment, the call will revert because that round doesn't exist this DOS will last for up to 24 hours depending on market settings. After the DOS ends, settingFunding will be able to catch up but it will now calculate the funding rate with different TWAP values.

## Impact

Loss of yield to a portion of all users in every market each time there is a phaseId shift

## Code Snippet

## Tool used

Manual Review

## Recommendation

I would recommend using a try block when calling the aggregator. If the roundID is nonzero and is reverting then the oracle needs try again with a lower phaseId

## Discussion

**asquare08**

This is a valid issue. We will fix it with post-mainnet releases because we are using trusted oracle (deployed on hubbleNet) with mainnet release and any updates to that will be notified prior.

# Issue M-11: User will be forced liquidated

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/234

## Found by

BugBusters

## Summary

The `addMargin` function have a vulnerability that can potentially lead to issues when the contract is paused. Users are unable to add margin during the paused state, which can leave them vulnerable to liquidation if their collateral value falls below the required threshold. Additionally, after the contract is unpaused, users can be subject to frontrunning, where their margin addition transactions can be exploited by other users.

## Vulnerability Detail

To better understand how this vulnerabilities could be exploited, let's consider a scenario:

1): The contract owner pauses the contract due to some unforeseen circumstances or for maintenance purposes.

2): During the paused state, User A wants to add margin to their account. However, they are unable to do so since the contract prohibits margin addition while paused.

3): Meanwhile, the price of the collateral supporting User A's account experiences significant fluctuations, causing the value of their collateral to fall below the required threshold for maintenance.

4): While the contract is still paused, User A's account becomes eligible for liquidation.

5): After some time, the contract owner decides to unpause the contract, allowing normal operations to resume.

6): User A tries to add margin to their account after the contract is unpaused. However, before their transaction is processed, User B, who has been monitoring the pending transactions, notices User A's margin addition transaction and quickly frontruns it by submitting a higher gas price transaction to liquidate User A's account instead.

Now userA will be forcefully liquidated even tho he wants to add the margin.

You can read more from this link

## Impact

The identified impact could be

1): Unfair Liquidations: Users can be unfairly liquidated if their margin addition transactions are frontrun by other users after the contract is unpaused. This can result in the loss of their collateral.

## Code Snippet

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/MarginAccount.sol#L136-L156

## Tool used

Manual Review

## Recommendation

Implement a Fair Liquidation Mechanism: Introduce a delay or waiting period before executing liquidation transactions. This waiting period should provide sufficient time for users to address their collateral issues or add margin.

## Discussion

**Nabeel-javaid**

escalate for 10 USDC

I think this issue should be considered as valid. The validity of issue should be accepted. The report clearly shows how a protocol owner actions (pause) will result in unfair liquidations causing loss of funds to users.

Also it is unlikely that on unpause human users will be able to secure their positions before MEV/liquidation bots capture the available profit. Hence the loss is certain.

For reference, similar issues were consider valid in the recent contests on sherlock, you can check it here and here. Maintaining a consistent valid/invalid classification standard will be ideal here.

**sherlock-admin2**

> escalate for 10 USDC
>
> I think this issue should be considered as valid. The validity of issue should be accepted. The report clearly shows how a protocol owner actions (pause) will result in unfair liquidations causing loss of funds to users.

> Also it is unlikely that on unpause human users will be able to secure their positions before MEV/liquidation bots capture the available profit. Hence the loss is certain.
>
> For reference, similar issues were consider valid in the recent contests on sherlock, you can check it here and here. Maintaining a consistent valid/invalid classification standard will be ideal here.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

**ctf-sec**

I mean if the liquidation cannot be paused then clearly a medium (there are similar past finding)

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/1f9a5ed0ca8f6004bbb7b099ecbb8ae796557849/hubble-protocol/contracts/MarginAccount.sol#L322

```
function liquidateExactRepay(address trader, uint repay, uint idx, uint
↪  minSeizeAmount) external whenNotPaused {
```

But The liquidation is paused as well

so recommend severity is low

a optional fix is adding a grace period for user to add margin when unpause the liquidation and addMargin

**Nabeel-javaid**

Hey @ctf-sec Adding more context, the problem is not that the liquidation functions can be paused or not, it is more of a front-running problem.

Liquidation functions cannot be paused individually, whole contract is paused which means that other functions with the whenNotPaused modifier cannot be accessed either.

So when the contract is un-paused, a user can be front-run before making the position healthy and is unfairly liquidated. For more reference see the following issue that are exactly the same .

Perrenial Contest Issue 190 Notional Contest Issue 203

I hope that added context will further help in judging.

**hrishibhat**

@ctf-sec I understand this is how it is design but there is a loss due front-run during unpause.

**ctf-sec**

https://github.com/sherlock-audit/2023-04-blueberry-judging/issues/118

seems like this issue has been historically judged as a medium

then a medium is fine

**hrishibhat**

Result: Medium Unique After further discussions with Lead Watson and Sponsor, considering this a valid medium based on the above comments

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:

- Nabeel-javaid: accepted

**asquare08**

This issue has more chances of occurrence when there are volatile assets as collateral. As we are launching with single collateral (USDC) initially, we'll fix this later.

# Issue M-12: No `minAnswer/maxAnswer` Circuit Breaker Checks while Querying Prices in Oracle.sol

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/241

## Found by

Bauchibred, BugBusters, Hama, crimson-rat-reach, ni8mare, rogue-lion-0619

## Summary

The Oracle.sol contract, while currently applying a safety check (this can be side stepped, check my other submission ) to ensure returned prices are greater than zero, which is commendable, as it effectively mitigates the risk of using negative prices, there should be an implementation to ensure the returned prices are not at the extreme boundaries (`minAnswer` and `maxAnswer`). Without such a mechanism, the contract could operate based on incorrect prices, which could lead to an over- or under-representation of the asset's value, potentially causing significant harm to the protocol.

## Vulnerability Details

Chainlink aggregators have a built in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value (i.e. LUNA crash) the price of the oracle will continue to return the minPrice instead of the actual price of the asset. This would allow user to continue borrowing with the asset but at the wrong price. This is exactly what happened to Venus on BSC when LUNA imploded. In its current form, the `getUnderlyingPrice()` function within the Oracle.sol contract retrieves the latest round data from Chainlink, if the asset's market price plummets below `minAnswer` or skyrockets above `maxAnswer`, the returned price will still be `minAnswer` or `maxAnswer`, respectively, rather than the actual market price. This could potentially lead to an exploitation scenario where the protocol interacts with the asset using incorrect price information.

Take a look at Oracle.sol#L106-L123:

```
function getLatestRoundData(AggregatorV3Interface _aggregator)
    internal
    view
    returns (
        uint80,
        uint256 finalPrice,
```

```
        uint256
    )
{
    (uint80 round, int256 latestPrice, , uint256 latestTimestamp, ) =
↪   _aggregator.latestRoundData();
    finalPrice = uint256(latestPrice);
    if (latestPrice <= 0) {
        requireEnoughHistory(round);
        (round, finalPrice, latestTimestamp) = getRoundData(_aggregator, round -
↪   1);
    }
    return (round, finalPrice, latestTimestamp);
}
```

## Illustration:

- Present price of TokenA is $10
- TokenA has a minimum price set at $1 on chainlink
- The actual price of TokenA dips to $0.10
- The aggregator continues to report $1 as the price.

Consequently, users can interact with protocol using TokenA as though it were still valued at $1, which is a tenfold overestimate of its real market value.

## Impact

The potential for misuse arises when the actual price of an asset drastically changes but the oracle continues to operate using the `minAnswer` or `maxAnswer` as the asset's price. In the case of it going under the `minAnswer` malicious actors obviously have the upperhand and could give their potential *going to zero* worth tokens to protocol

## Code Snippet

PriceOracle.sol#L60-L72

## Tool used

Manual Audit

## Recommendation

Since there is going to be a whitelist of tokens to be added, the minPrice/maxPrice could be checked and a revert could be made when this is returned by chainlink or

a fallback oracle that does not have circuit breakers could be implemented in that case

## Discussion

**asquare08**

This is a valid concern. But we will fix this in later releases as initially, we are launching with blue chip tokens only and single collateral (USDC).

**asquare08**

As mentioned in this comment, we'll fix this later.

# Issue M-13: Potential accounting problems due to issue in `ClearingHouse.updatePositions()`

Source: https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/248

## Found by

lemonmon

## Summary

Potential issue in `ClearingHouse.updatePositions()` when lastFundingTime is not being updated by `ClearingHouse.settleFunding`.

## Vulnerability Detail

`ClearingHouse.lastFundingTime` is only updated, when `_nextFundingTime` is not zero:

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/ClearingHouse.sol#L281-L282

`_nextFundingTime` is determined a few lines above by a call to `amms[i].settleFunding()`:

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/ClearingHouse.sol#L267

The return value of `amms[i].settleFunding()` can be zero for `_nextFundingTime`, if the `block.timestamp` is smaller than the next funding time:

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/AMM.sol#L236-L249

That means that if the last market inside the `amms` array has not reached the next funding time at `block.timestamp`, `_nextFundingTime` will be zero and `lastFundingTime` will not be updated.

Then when `ClearingHouse.updatePositions()` is called, it will not process `fundingPayment` because `lastFundingTime` was not updated:

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/ClearingHouse.sol#L241-L250

## Impact

Unrealized funding payments are not settled correctly, potentially leading to incorrect margin accounting when opening a position. Also

`marginAccount.realizePnL()` (line 255 ClearingHouse.sol) won't get called, so the trader won't receive funds that they should receive.

Note: `ClearingHouse.updatePositions()` is called by `ClearingHouse._openPosition` (line 141 ClearingHouse.sol).

Note: `ClearingHouse.liquidate` -> `ClearingHouse.openPosition` -> `ClearingHouse._openPosition`

There can be multiple potential issues with accounting that can result due to this issue, both when liquidating and when opening a position.

## Code Snippet

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/ClearingHouse.sol#L281-L282

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/ClearingHouse.sol#L267

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/AMM.sol#L236-L249

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/ClearingHouse.sol#L241-L250

https://github.com/sherlock-audit/2023-04-hubble-exchange/blob/main/hubble-protocol/contracts/ClearingHouse.sol#L140-L141

## Tool used

Manual Review

## Recommendation

Consider adjusting the code inside `ClearingHouse.settleFunding()` to account for the case where the last market inside the `amms` array returns zero for `_nextFundingTime` when `Amm.settleFunding()` is called (line 267 ClearingHouse.sol). For example introduce a boolean variable that tracks whether a market inside `amms` array didn't return zero for `_nextFundingTime`.

```
// ClearingHouse
// settleFunding
268             if (_nextFundingTime != 0) {
269                 _marketReachedNextFundingTime = true; // <-- @audit new
↪  boolean to track

281         if (_marketReachedNextFundingTime) {
```

```
282             lastFundingTime = _blockTimestamp();
283         }
```

## Discussion

### asquare08

All amms will have the same `nextFundingTime` as `amm.settleFunding` is called together for all amms in a single tx. So _nextFundingTime will either be 0 or none for all amms.

```
    function settleFunding() override external onlyDefaultOrderBook {
        uint numAmms = amms.length;
        uint _nextFundingTime;
        for (uint i; i < numAmms; ++i) {
            int _premiumFraction;
            int _underlyingPrice;
            int _cumulativePremiumFraction;
            (_premiumFraction, _underlyingPrice, _cumulativePremiumFraction,
↪   _nextFundingTime) = amms[i].settleFunding();
            if (_nextFundingTime != 0) {
                emit FundingRateUpdated(
                    i,
                    _premiumFraction,
                    _underlyingPrice.toUint256(),
                    _cumulativePremiumFraction,
                    _nextFundingTime,
                    _blockTimestamp(),
                    block.number
                );
            }
        }
        // nextFundingTime will be same for all amms
        if (_nextFundingTime != 0) {
            lastFundingTime = _blockTimestamp();
        }
    }



**ctf-sec**

Closed this issue based on sponsor's comment,

the watson is welcome to escalate with a more clear impact and more proof

**lemonmon1984**
```

```
Escalate
> All amms will have the same nextFundingTime as amm.settleFunding is called
 ↪  together for all amms in a single tx.

It is not neccessarily true for certain conditions, in which
 ↪  ClearingHouse.whitelistAmm() is involved.
One possible scenario would be:
1. A couple of AMMs are whitelisted via ClearingHouse.whitelistAmm() where a
 ↪  nextFundingTime is assigned to them.
1. After the nextFundingTime was reached, ClearingHouse.whitelistAmm() is called
 ↪  again to add another AMM.
1. Then the ClearingHouse.settleFunding() is called and returns 0 for the last
 ↪  AMM that was just added in the previous step, because its nextFundingTime is
 ↪  in the future and ClearingHouse.sol line 248 is true and returns 0 for
 ↪  nextFundingTime, which then leads to the issue described here.

Note: ClearingHouse.whitelistAmm() and ClearingHouse.settleFunding() is likely
 ↪  to be initiated by separate entities. Therefore, the order of transactions
 ↪  is not guaranteed. Also the tx execution can be delayed due to network
 ↪  congestions. That's why this scenario may occur.

**Example:**

fundingPeriod: 3h
call ClearingHouse.whitelistAmm():  at 16:00 (block.timestamp)

then AMM.startFunding()  gets called l521 in ClearingHouse.sol
    here it calculates:
        nextFundingTime = ((16 + 3) / 3) * 3 = 18


at 19:00 another AMM is added:
    call ClearingHouse.whitelistAmm():  at 19:00 (block.timestamp)
    then AMM.startFunding()  gets called l521 in ClearingHouse.sol
        here it calculates:
            nextFundingTime = ((19 + 3) / 3) * 3 = 21

then also at 19:00 after another AMM was just added:
    call ClearingHouse.settleFunding()
        which calls AMM.settleFunding, and the last index of AMMs contains the
 ↪  added AMM from 19:00 with nextFundingTime 21
            so AMM.settleFunding will return 0,0,0,0 l249 because
 ↪  nextFundingTime is in the future.
                This leads to exactly the issue described that then on line 281
 ↪  in ClearingHouse.sol _nextFundingTime will be 0.
```



```
### Other scenario
In another scenario, if the ClearingHouse.settleFunding() is called past the
↪  nextFundingTime due to network congestion,
the nextFundingTime will shift from the fixed schedule. While it is shifted, if
↪  the ClearingHouse.whitelistAmm() is called, then it gives a certain window
↪  for the ClearingHouse.settleFunding() call will face the same issue.

**sherlock-admin2**

 > Escalate
> > All amms will have the same nextFundingTime as amm.settleFunding is called
↪  together for all amms in a single tx.
>
> It is not neccessarily true for certain conditions, in which
↪  ClearingHouse.whitelistAmm() is involved.
> One possible scenario would be:
> 1. A couple of AMMs are whitelisted via ClearingHouse.whitelistAmm() where a
↪  nextFundingTime is assigned to them.
> 1. After the nextFundingTime was reached, ClearingHouse.whitelistAmm() is
↪  called again to add another AMM.
> 1. Then the ClearingHouse.settleFunding() is called and returns 0 for the last
↪  AMM that was just added in the previous step, because its nextFundingTime is
↪  in the future and ClearingHouse.sol line 248 is true and returns 0 for
↪  nextFundingTime, which then leads to the issue described here.
>
> Note: ClearingHouse.whitelistAmm() and ClearingHouse.settleFunding() is likely
↪  to be initiated by separate entities. Therefore, the order of transactions
↪  is not guaranteed. Also the tx execution can be delayed due to network
↪  congestions. That's why this scenario may occur.
>
> **Example:**
>
> fundingPeriod: 3h
> call ClearingHouse.whitelistAmm():  at 16:00 (block.timestamp)
>
> then AMM.startFunding()  gets called l521 in ClearingHouse.sol
>     here it calculates:
>         nextFundingTime = ((16 + 3) / 3) * 3 = 18
>
>
> at 19:00 another AMM is added:
>     call ClearingHouse.whitelistAmm():  at 19:00 (block.timestamp)
>     then AMM.startFunding()  gets called l521 in ClearingHouse.sol
>         here it calculates:
>             nextFundingTime = ((19 + 3) / 3) * 3 = 21
>
> then also at 19:00 after another AMM was just added:
>     call ClearingHouse.settleFunding()
```



> which calls AMM.settleFunding, and the last index of AMMs contains the added AMM from 19:00 with nextFundingTime 21
> so AMM.settleFunding will return 0,0,0,0 l249 because nextFundingTime is in the future.
> This leads to exactly the issue described that then on line 281 in ClearingHouse.sol _nextFundingTime will be 0.
>
>
> ### Other scenario
> In another scenario, if the ClearingHouse.settleFunding() is called past the nextFundingTime due to network congestion,
> the nextFundingTime will shift from the fixed schedule. While it is shifted, if the ClearingHouse.whitelistAmm() is called, then it gives a certain window for the ClearingHouse.settleFunding() call will face the same issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

**asquare08**

A good corner case is described. It's a valid issue and can be marked as medium severity

**IAm0x52**

Agreed with escalation, valid corner case

**hrishibhat**

Result:
Medium
Unique

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:
- [lemonmon1984](https://github.com/sherlock-audit/2023-04-hubble-exchange-judging/issues/248/#issuecomment-1642709772): accepted

**asquare08**

```
Whitelist amm is governance function, we'll take care that when whitelisting new
↪  amm, all amms should have same `nextFundingTime`
```