Hubble contest
Findings & Analysis Report

2022-06-02

Table of contents

- Overview
    - About C4
    - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (3)
    - [H-01] Update initializer modifier to prevent reentrancy during initialization
    - [H-02] denial of service
    - [H-03] InsuranceFund depositors can be priced out & deposits can be stolen
- Medium Risk Findings (17)
    - [M-01] Liquidations can be run on the bogus Oracle prices
    - [M-02] Hidden governance
    - [M-03] ClearingHouse May Whitelist Duplicate AMMs
    - [M-04] `settleFunding` will exceed block gas with more markets and activity

  - [M-05] `Oracle.getUnderlyingPrice` could have wrong decimals
  - [M-06] After debt seizure from `InsuranceFund`, user can dilute all past participants.
  - [M-07] ClearingHouse margin calculations will break up if an AMM returning non-6 decimals positions be white listed
  - [M-08] All AMMs have to be past nextFundingTime to update
  - [M-09] Ownership of `Swap.vy` cannot be transferred
  - [M-10] Blocking of the VUSD withdrawals is possible if the reserve token doesn't support zero value transfers
  - [M-11] Users are able to front-run bad debt settlements to avoid insurance costs
  - [M-12] AMM Cannot Be `initialize()` Except By Governance
  - [M-13] Assets sent from `MarginAccount` to `InsuranceFund` will be locked forever
  - [M-14] Liquidation is vulnerable to sandwich attacks
  - [M-15] [WP-H7] `InsuranceFund#syncDeps()` may cause users' fund loss
  - [M-16] USDC blacklisted accounts can DoS the withdrawal system
  - [M-17] Usage of an incorrect version of Ownable library can potentially malfunction all `onlyOwner` functions
- Low Risk and Non-Critical Issues
  - L-01 PREVENT DIV BY 0
  - L-02 Single-step change of governance address is extremely risky
  - L-03 Front-runnable Initializers
  - L-04 Incompatibility With Rebasing/Deflationary/Inflationary tokens
  - L-05 Missing zero-address check in constructors and the setter functions
  - L-06 Missing events for governor only functions that change critical parameters
  - L-07 Deprecated safeApprove() function
  - L-08 The Contract Should Approve(0) first

- L-09 Missing Pause Modifier On the InsuranceFunds contract
- N-01 Use of Block.timestamp
- N-02 Missing Re-entrancy Guard
- Gas Optimizations
  - Table of Contents
  - Foreword
  - Summary
  - File: AMM.sol
  - File: ClearingHouse.sol
  - File: InsuranceFund.sol
  - File: Oracle.sol
  - File: Interfaces.sol
  - File: MarginAccount.sol
  - File: VUSD.sol
  - General Recommendations
- Disclosures

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Hubble smart contract system written in Solidity. The audit contest took place between February 17—February 23 2022.

Wardens

45 Wardens contributed reports to the Hubble contest:

1. Dravee
2. cmichel
3. kirk-baird
4. hyh
5. 0xliumin
6. danb
7. WatchPug (jtp and ming)
8. robee
9. gzeon
10. throttle
11. 0x1f8b
12. leastwood
13. itsmeSTYJ
14. defsec
15. csanuragjain
16. Omik
17. minhquanym
18. Ruhum
19. Meta0xNull
20. IllIllI
21. pauliax
22. rfa
23. ye0lde
24. kenta
25. bw
26. hubble (ksk2345 and shri4net)

27. cccz
28. 0v3rf10w
29. sorrynotsorry
30. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
31. Jujic
32. bobi
33. peritoflores
34. 0xwags
35. 0x0x0x
36. jayjonah8
37. Nikolay
38. d4rk
39. Tomio

This contest was judged by the Float Capital team: moose-code and JasoonS.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 17 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 30 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 21 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Hubble contest repository, and is composed of 7 smart contracts written in the Solidity programming language

(2,109 lines of code) as well as 3 contracts written in the Vyper programming language (1,561 lines of code).

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on [OWASP standards](#).

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

High Risk Findings (3)

[H-01] Update initializer modifier to prevent reentrancy during initialization

Submitted by Dravee

https://github.com/code-423n4/2022-02-hubble/blob/main/package.json#L17
https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/legos/Governable.sol#L5
https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/legos/Governable.sol#L24

While Governable.sol is out of scope, I figured this issue would still be fair game.

The solution uses: `"@openzeppelin/contracts": "4.2.0"` .

This dependency has a known high severity vulnerability:

https://security.snyk.io/vuln/SNYK-JS-OPENZEPPELINCONTRACTS-2320176

Which makes this contract vulnerable:

```
File: Governable.sol
05: import { Initializable } from "@openzeppelin/contracts/proxy/util
...
24: contract Governable is VanillaGovernable, Initializable {}
```

This contract is inherited at multiple places:

```
contracts/AMM.sol:
  11: contract AMM is IAMM, Governable {

contracts/InsuranceFund.sol:
  13: contract InsuranceFund is VanillaGovernable, ERC20Upgradeable {

contracts/Oracle.sol:
  11: contract Oracle is Governable {

contracts/legos/HubbleBase.sol:
  15: contract HubbleBase is Governable, Pausable, ERC2771Context {

contracts/ClearingHouse.sol:
  11: contract ClearingHouse is IClearingHouse, HubbleBase {

contracts/MarginAccount.sol:
  25: contract MarginAccount is IMarginAccount, HubbleBase {
```

ìnitializer()`is used here:

```
contracts/AMM.sol:
  99:     ) external initializer {

contracts/ClearingHouse.sol:
  44:     ) external initializer {
```

```
contracts/MarginAccount.sol:
  124:     ) external initializer {

contracts/Oracle.sol:
  20:     function initialize(address _governance) external initializ
```

Recommended Mitigation Steps

Upgrade `@openzeppelin/contracts` to version 4.4.1 or higher.

[atvanguard (Hubble) confirmed and resolved]

[moose-code (judge) commented]:

Agreed. Other issues such as [this] have also popped up, so always safest to be on the newest OZ. This includes for contracts and contracts-upgradeable packages.

[H-02] denial of service

Submitted by danb, also found by cmichel, csanuragjain, hyh, kirk-baird, leastwood, MetaOxNull, minhquanym, Omik, robee, Ruhum, and throttle

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/VUSD.sol#L53

processWithdrawals can process limited amount in each call.
An attacker can push to withdrawals enormous amount of withdrawals with amount = 0.
In order to stop the dos attack and process the withdrawal, the governance needs to spend as much gas as the attacker.
If the governance doesn't have enough money to pay for the gas, the withdrawals can't be processed.

Proof of Concept

Alice wants to attack vusd, she spends 1 millions dollars for gas to push as many withdrawals of amount = 0 as she can.
If the governance wants to process the deposits after Alices empty deposits, they

also need to spend at least 1 million dollars for gas in order to process Alice's withdrawals first.
But the governance doesn't have 1 million dollars so the funds will be locked.

Recommended Mitigation Steps

Set a minimum amount of withdrawal. e.g. 1 dollar

```
function withdraw(uint amount) external {
    require(amount >= 10 ** 6);
    burn(amount);
    withdrawals.push(Withdrawal(msg.sender, amount));
}
```

atvanguard (Hubble) confirmed, but disagreed with High severity and commented:

Confirming this is an issue. Would classify it as `2 (Med Risk)` because this attack is expensive to carry out.

atvanguard (Hubble) resolved

moose-code (judge) commented:

Would be interested to see the exact gas cost of executing withdraw. The thing is the grievance costs only gas to execute and the withdraw function is relatively cheap from first glance. The main issue here is that it can become SUPER expensive to clear the que in gas. I.e. if the attacker builds up a que of 200 withdrawals, some unknowning sucker is going to pay for more than 200 erc20 transfers in order to get their money out. Thats more than anyone would want to pay, and further since so much gas limit would be needed for this to be executed, to fit into a block you are going to have to pay a huge price.

So basically it costs attacker x to execute, which means it is also going to cost next user likely even more than x to fix the problem.

Also the que is not cleared so processWithdrawals becomes a really expensive function. If the items were cleared and set back to zero it would make it less expensive to de-que the que.

This being said we definitely have this at at least medium severity. $10k in gas to constantly brick users withdrawls from protocol for a week is a serious issue and not the biggest cost for an attack.

@JasoonS, going to put this as medium. Let's discuss whether we want to have it as high.
moose-code (judge) commented:

Okay, going to keep this as high severity. The cost to fix the attack can be more than what the attack costs in total. It also burdens a random unsuspecting user with a really high gas cost to try and get their withdrawal. There are many good suggestions on how to fix this.

[H-03] InsuranceFund depositors can be priced out & deposits can be stolen

Submitted by cmichel, also found by danb

https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/InsuranceFund.sol#L44-L54

The `InsuranceFund.deposit` function mints initial `shares` equal to the deposited amount.
The deposit / withdraw functions also use the VUSD contract balance for the shares computation. (`balance() = vusd.balanceOf(address(this))`)

It's possible to increase the share price to very high amounts and price out smaller depositors.

Proof of Concept

- `deposit(_amount = 1)` : Deposit the smallest unit of VUSD as the first depositor. Mint 1 share and set the total supply and VUSD balance to `1` .

- Perform a direct transfer of `1000.0` VUSD to the `InsuranceFund`. The `balance()` is now `1000e6 + 1`
- Doing any deposits of less than `1000.0` VUSD will mint zero shares: `shares = _amount * _totalSupply / _pool = 1000e6 * 1 / (1000e6 + 1) = 0`.
- The attacker can call `withdraw(1)` to burn their single share and receive the entire pool balance, making a profit. (`balance() * _shares / totalSupply() = balance()`)

I give this a high severity as the same concept can be used to always steal the initial insurance fund deposit by frontrunning it and doing the above-mentioned steps, just sending the frontrunned deposit amount to the contract instead of the fixed `1000.0`. They can then even repeat the steps to always frontrun and steal any deposits.

Recommended Mitigation Steps

The way [UniswapV2 prevents this] is by requiring a minimum deposit amount and sending `1000` initial shares to the zero address to make this attack more expensive. The same mitigation can be done here.

[atvanguard (Hubble) confirmed]

Medium Risk Findings (17)

[M-01] Liquidations can be run on the bogus Oracle prices

Submitted by hyh, also found by 0x1f8b, cccz, csanuragjain, defsec, hubble, leastwood, pauliax, WatchPug, and ye0lde

If the price feed is manipulated in any way or there is any malfunction based volatility on the market, a malicious user can use this to liquidate a healthy position.

An attacker can setup a monitoring of the used Oracle feed and act on observing a price outbreak (for example, zero price, which is usually a subject to filtration), liquidating the trader position which is perfectly healthy otherwise, obtaining the collateral with a substantial discount at the expense of the trader.

The same is for a flash crash kind of scenario, i.e. a price outbreak of any nature will allow for non-market liquidation by an attacker, who has the incentives to setup such a monitoring and act on such an outbreak, knowing that it will not be smoothed or filtered out, allowing a liquidation at a non-market price that happen to be printed in the Oracle feed

Proof of Concept

Oracle.getUnderlyingPrice just passes on the latest Oracle answer, not checking it anyhow:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/Oracle.sol#L24-L35

It is then used in liquidation triggers providing isLiquidatable and _getLiquidationInfo functions:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/MarginAccount.sol#L249

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/MarginAccount.sol#L465

Recommended Mitigation Steps

Add a non-zero Oracle price check, possibly add an additional Oracle feed information usage to control that the price is fresh. Please consult the Chainlink for that as OCR introduction might have changed the state of the art approach (i.e. whether and how to use latestRoundData returned data):

https://docs.chain.link/docs/off-chain-reporting/

Regarding any price spikes it is straightforward to construct a mitigation mechanics for such cases, so the system will be affected by sustainable price movements only.

As price outrages provide a substantial attack surface for the project it’s worth adding some complexity to the implementation.

One of the approaches is to track both current and TWAP prices, and condition all state changing actions, including liquidations, on the current price being within a threshold of the TWAP one. If the liquidation margin level is conservative enough and TWAP window is small enough this is safe for the overall stability of the system, while providing substantial mitigation mechanics by allowing state changes on the locally calm market only.

Another approach is to introduce time delay between liquidation request and actual liquidation. Again, conservative enough margin level plus small enough delay keeps the system safe, while requiring that market conditions allow for liquidation both at request time and at execution time provides ample filtration against price feed outbreaks

atvanguard (Hubble) confirmed

moose-code (judge) decreased severity from High to Medium

[M-02] Hidden governance

Submitted by 0x1f8b

The contract use two governance model, one looks hidden.

Proof of Concept

The VUSD contract uses `VanillaGovernable` but inherits from `ERC20PresetMinterPauserUpgradeable` and this contract uses roles to use some administrative methods like `pause` or `mint`.

This two-governance model does not seem necessary and can hide or raise suspicion about a rogue pool, thus damaging the user's trust.

Recommended Mitigation Steps

Unify governance in only one, VanillaGovernable or role based.

atvanguard (Hubble) confirmed and resolved

moose-code (judge) commented:

Yes, a good suggestion to keep governance more tightly coupled. OZ has AccessControlledAndUpgradeable which is really nice. Various roles for varying level of admin functionality. Allows tighter controls on more controversial items and easier control on less controversial items.

[M-03] ClearingHouse May Whitelist Duplicate AMMs

Submitted by kirk-baird

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L339-L342
https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L269-L282

`ClearingHouse.sol` allows the Governance protocol to whitelist `AMM.sol` contracts. These contracts allow users to earn profits based on the price of a base asset against a quote asset.

It is possible to add the same `AMM` twice in the function `whitelistAmm()`. The impact is that unrealized profits will be counted multiple times. As a result the liquidation calculations will be incorrect, potentially allowing users to trade while insolvent or incorrectly liquidating solvent users.

Note `whitelistAmm()` may only be called by Governance.

Proof of Concept

The function `getTotalNotionalPositionAndUnrealizedPnl()` will iterate over all `amms` summing the `unrealizedPnl` and `notinoalPosition`, thus if an `amm` is repeated the `unrealizedPnl` and `notionalPosition` of that asset will be counted multiple times.

This is used in `_calcMarginFraction()` which calculates a users margin as a fraction of the total position. The margin fraction is used to determine if a user is liquitable or is allowed to open new positions.

Recommended Mitigation Steps

Consider ensuring the `AMM` does not already exist in the list when adding a new `AMM`.

```
function whitelistAmm(address _amm) external onlyGovernance {
    for (uint256 i; i < amm.length; i++) {
        require(amm[i] != IAMM(_amm), "AMM already whitelisted");
    }
    emit MarketAdded(amms.length, _amm);
    amms.push(IAMM(_amm));
}
```

[atvanguard (Hubble) confirmed, but disagreed with Medium severity and commented]:

As mentioned in [#40], the system relies on the admin to do the right thing; hence disagreeing with the severity. Still, it's a good idea to have this check.

[atvanguard (Hubble) resolved]

[moose-code (judge) commented]:

Practical advice that prevents a catastrophic issue that could very possibly occur (having run deployment / whitelist and many other scripts, it's way too easy to run something again etc and end up in this situation - even though it feels like it would never be possible).

[M-04] `settleFunding` will exceed block gas with more markets and activity

Submitted by bw, also found by cmichel, Dravee, gzeon, Omik, and Ruhum

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L129
https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/AMM.sol#L678

As the number of supported markets grow, `settleFunding` will reach a point were it exceeds the block gas limit on Avalanche C-Chain. This will prevent users from calling the function and cause a wide spread Denial of Service.

Looking at transactions for the current testnet deployment, `settleFunding` already reaches almost 10% of the block gas limit. This is due settle funding iteratively looping through each market, with each iteration entering an unbounded `while` loop in `_calcTwap`. The more active the markets are, the more gas intensive `_calcTwap` becomes, as more snapshots need to be traversed.

The combination of more active markets and an increase in available markets make it very likely that some users will be unable to call `settleFunding` in the long run.

Proof of Concept

Example of transactions on testnet:

Gas	Limit%	Link
658428	8.2%	https://testnet.snowtrace.io/tx/0x8123a5658a98e694e7428e66c9e5f9d5cbff8af93d543ed51a80cb367bcccd2c
653810	8.1%	https://testnet.snowtrace.io/tx/0xf126eb05245580a73981228d6f0f8d607ad038ca0b68593f0c903e210c1c2c57

Recommended Mitigation Steps

Users should be allowed to settle funding per market or using an array of markets opposed to all markets at once.

```
function settleFundingForMarkets(IAMM[] markets) override external wh
    for (uint i = 0; i < markets.length; i++) {
        markets[i].settleFunding();
    }
}
```

In this way the gas cost will not increase with the number of markets created over time.

atvanguard (Hubble) acknowledged, but disagreed with High severity and commented:

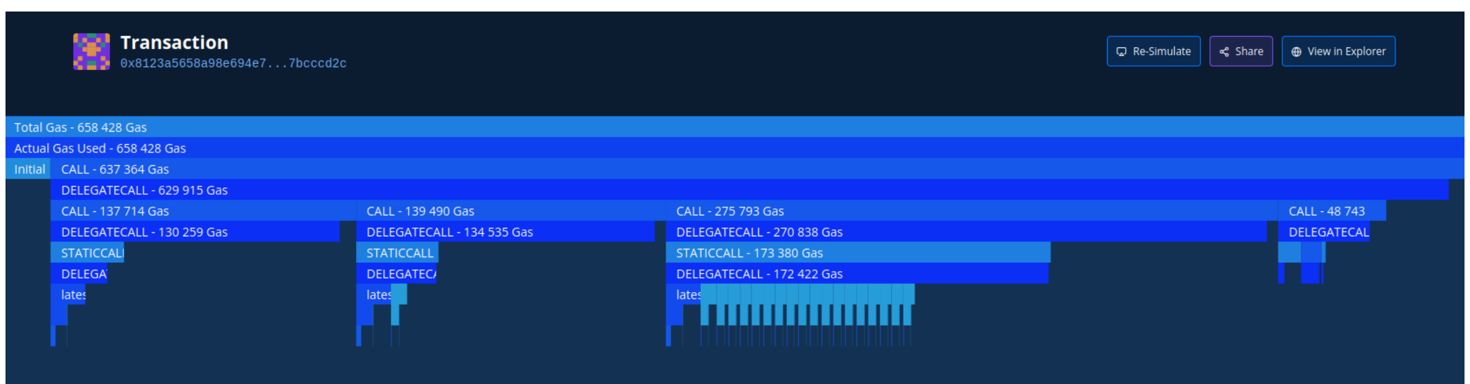
It’s a known issue that adding many more markets will eventually exceed block gas limit on many operations (not just `settleFunding`). For that reason, DAO governance has to be careful with not adding too many markets. Would classify this as `0 (Informational)`.

moose-code (judge) commented:

This is definitely more than informational. You can see in the gas profiler on only a limited amount of markets with not much action, lots of gas was used. This has potential to be a much bigger issue in the future.

Very rightly so the warden points out `_calcTwap` as intensive which is shown on the profiler below for the one market. Calculating a TWAP across many different markets could blow this function up very quickly.

Further more, if you take ALL the gas in a single block, this becomes really expensive and difficult as you need to push out a lot of other high priority transactions that are pending. If this was needing to be executed during an NFT minting spree it would be tough.



@JasoonS, let’s discuss on medium vs high for this one.

moose-code (judge) decreased severity to Medium and commented:

Going to have this as medium.

[M-05] `Oracle.getUnderlyingPrice` could have wrong decimals

Submitted by cmichel

https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/Oracle.sol#L34

The `Oracle.getUnderlyingPrice` function divides the chainlink price by `100`.
It probably assumes that the answer for the underlying is in 8 decimals but then wants to reduce it for 6 decimals to match USDC.

However, arbitrary `underlying` tokens are used and the chainlink oracles can have different decimals.

Recommended Mitigation Steps

While most USD price feeds use 8 decimals, it's better to take the on-chain reported decimals into account by doing `AggregatorV3Interface(chainLinkAggregatorMap[underlying]).decimals()`, see [Chainlink docs](#).
The price should then be scaled down to 6 decimals.

[atvanguard (Hubble) confirmed, but disagreed with High severity and commented](#):

All chainlink USD pairings are expected to have 8 decimals hence disagreeing with severity; but yes agree that asserting this check when adding a new asset is a good idea.

[moose-code (judge) decreased severity to Medium and commented](#):

Downgrading to medium. Dividing by magic numbers (100) should clearly comment assumptions.

[M-06] After debt seizure from `InsuranceFund`, user can dilute all past participants.

Submitted by 0xliumin

https://github.com/code-423n4/2022-02-hubble/blob/ed1d885d5dbc2eae24e43c3ecbf291a0f5a52765/contracts/InsuranceFund.sol#L56

A user can get a much larger portion of the pool as it recovers from a debt seizure. The intent of the insurance pool seems to be that it could recover from a bad debt event.

Proof of Concept

1. Alice is the first LP to the insurance pool, and deposits 1e18 shares.
2. `seizeBadDebt` is called with 2e18. Now, there are `pendingObligations = 1e18`, and there is 0 vusd in the insurance fund.
3. Bob (the attacker) directly transfers 1e18 + 1 vUSD.
4. Bob calls deposit with 1e18 vUSD. All pending obligations will be settled, but there will only be 1 vUSD left in the pool before Bob's deposit. Bob receives `shares = 1e18 * 1e18 / 1`. As a result, Bob will get `1e36` shares, diluting Alice's share of the pool. Bob will be able to take a much larger share of all future profits from the insurance fund until more bad debt is seized. Bob only provided 2e18 + 1 liqudiity, but received an exponentially larger number of shares than Alice.

Recommended Mitigation Steps

It depends on how you want this to work. You could keep track of the total amount ever contributed by users, and use that for calculations. Or just make staking 1 vUSD = 1 share if the pool total is below the total number of shares.

atvanguard (Hubble) disputed and commented:

Disputing this. It is by design. LPs who were in the insurance fund will be burnt during a bad debt settlement.

JasoonS (judge) decreased severity to Medium

[M-07] ClearingHouse margin calculations will break up if an AMM returning non-6 decimals positions be white listed

Submitted by hyh

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L332

It is assumed that VAMM returned positions have exactly `6` decimals for all AMMs white listed in ClearingHouse.

In the same time an array of different AMMs/VAMMs is supported, and there are no guarantees/checks of the precision of the position values they return.

If an VAMM that have different precision is whitelisted, for example having 18 decimals for position figures, then margin requirements checks become invalid.

This will lead to various malfunctions, say perfectly valid positions will be liquidated by any attacker noticing that the calculations are skewed.

Proof of Concept

ClearingHouse's _calcMarginFraction is the function that is used for margin requirements checks:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L163-L167

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L188-L189

_calcMarginFraction calls getNotionalPositionAndMargin:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L319-L320

getNotionalPositionAndMargin calls getTotalNotionalPositionAndUnrealizedPnl:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L291

getTotalNotionalPositionAndUnrealizedPnl sums up AMM's getNotionalPositionAndUnrealizedPnl results:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L269-L282

AMM's getNotionalPositionAndUnrealizedPnl returns vamm.get_notional result:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/AMM.sol#L395-L410

The above calls are linear decimals wise (i.e. do subtractions/additions kind of operations, preserving the decimals).

Then, _getMarginFraction mixes up these notionalPosition and margin, obtained from AMM without rescaling, as if they are PRECISION scaled:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L332

PRECISION is hard coded to be `1e6` :

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/ClearingHouse.sol#L15

For other VAMM operations base precision is set to `1e18` :

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/AMM.sol#L17

For example, VAMM returned supply is assumed to have 18 decimals:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/AMM.sol#L523

Comment says that exchangeExactOut returned quantity will have 6 decimals precision:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/AMM.sol#L495

As the system imply that VAMMs can vary it is neither guaranteed, nor checked in any way (briefly checked dydx api code, it looks like there are no explicit guarantees either).

If any of VAMM referenced via white listed AMMs return VAMM.get*notional with decimals different from* `6` *, the* \calcMarginFraction result will become grossly incorrect.

Recommended Mitigation Steps

If AMM contract is desired to deal with various VAMMs, consider removing decimals related hard coding, adding decimals variables and scaling VAMM returned results accordingly, so that position and margin values' decimals of 6, implied by ClearingHouse logic, be ensured.

atvanguard (Hubble) disputed and commented:

Protocols developers will ensure that correct decimals are used everywhere. It's not possible to assert this in code at all places.

JasoonS (judge) decreased severity to Medium and commented:

Set to medium as unlikely that this would be done. However, checks and notices in the documentation of this code would be very important to prevent new devs from making these mistakes.

[M-08] All AMMs have to be past nextFundingTime to update

Submitted by 0xliumin

https://github.com/code-423n4/2022-02-hubble/blob/ed1d885d5dbc2eae24e43c3ecbf291a0f5a52765/contracts/AMM.sol

#L348

settleFunding calls will revert until all AMMs are ready to be updated.

Proof of Concept

AMM 1 has a nextFundingTime of now. AMM 2 has a nextFundingTime in 30 minutes. AMM 1 won't be able to be updated until after AMM 2's nextFundingTime elapses.

Recommended Mitigation Steps

You shouldn't revert at the place mentioned in the links to affected code. Just return so that the other AMMs can still get updated.

atvanguard (Hubble) confirmed and resolved

[M-09] Ownership of `Swap.vy` cannot be transferred

Submitted by gzeon

Ownership transfer function of Swap.vy is commented out. Fund can be stuck if an AMM and governance change/upgrade is required.

Proof of Concept

https://github.com/code-423n4/2022-02-hubble/blob/ed1d885d5dbc2eae24e43c3ecbf291a0f5a52765/contracts/curve-v2/Swap.vy#L1129

atvanguard (Hubble) disputed and commented:

Intended behavior because we don't use `self.owner` after the initial setup.

JasoonS (judge) commented:

If that is the case, then shouldn't this function have a check that the AMM can't be set multiple times? https://github.com/code-423n4/2022-02-hubble/blob/ed1d885d5dbc2eae24e43c3ecbf291a0f5a52765/contracts/curve-v2/Swap.vy#L966-L970

There is risk if the owner keys get compromised - also there is no progressive security if you can't change this.

IE it could start as an EOA - and progress to a multisig owner etc.

Leaving at medium severity - if you have an owner, there should always be a way to update it as to improve the security (and potentially decentralization) of the system over time.

[M-10] Blocking of the VUSD withdrawals is possible if the reserve token doesn't support zero value transfers

Submitted by hyh

VUSD withdraw queue will be blocked and user funds frozen simply by requesting a zero value withdraw, if the reserve token doesn't support zero value transfers.

Putting it medium only on an assumption that reserve will be USDC and the probability is low, but VUSD do allow any reserve token and the impact here is both funds freeze and stopping of the operations

Proof of Concept

It is possible to burn zero amount in OZ implementation:

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/token/ERC20/ERC20Upgradeable.sol#L285-L300

So, withdraw will burn zero amount and put it to the queue:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/VUSD.sol#L48

USDC does support zero value transfers, but not all the tokens do:

https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers

Currently VUSD can use any reserve token:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/VUSD.sol#L33

Withdraw queue position can be modified in the `processWithdrawals` function only.

But it will fail every time on the zero amount entry, as there is no way to skip it (and mint VUSD back, for example), so anything else after this zero entry will not be processed:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/VUSD.sol#L62

This way the withdrawal functionality and the corresponding user funds will be frozen within VUSD contract, which will become inoperable

Recommended Mitigation Steps

Consider adding a zero amount check, as it doesn't cost much, while zero transfer doesn't make sense anyway.

Now:

```
reserveToken.safeTransfer(withdrawal.usr, withdrawal.amount);
reserve -= withdrawal.amount;
```

To be:

```
if (withdrawal.amount > 0) {
        reserveToken.safeTransfer(withdrawal.usr, withdrawal.amount);
        reserve -= withdrawal.amount;
}
```

atvanguard (Hubble) disputed and commented:

Not an issue because reserveToken is intended to be USDC.

[JasoonS (judge) commented](#):

Not specified in spec for the audit. Giving to submitter.

[M-11] Users are able to front-run bad debt settlements to avoid insurance costs

Submitted by kirk-baird, also found by itsmeSTYJ

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/InsuranceFund.sol#L71-L75
https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/InsuranceFund.sol#L62-L69

A user is able to front-run the call to `seizeBadDebt()` in `InsuranceFund.sol` to avoid paying the insurance costs.

`seizeBadDebt()` is called by `MarginAccount.settleBadDebt()` which is a public function. When this functions is called the transaction will appear in the mem pool. A user may then call `InsuranceFund.withdraw()` to withdraw all of their shares. If they do this with a higher gas fee it will likely be processed before the `settleBadDebt()` transaction. In this way they will avoid incurring any cost from the assets being seized.

The impact is that users may gain their share of the insurance funding payments with minimal risk (minimal as there is a change the front-run will not succeed) of having to repay these costs.

Proof of Concept

```
    function withdraw(uint _shares) external {
        settlePendingObligation();
        require(pendingObligation == 0, "IF.withdraw.pending_obligati
        uint amount = balance() * _shares / totalSupply();
        _burn(msg.sender, _shares);
        vusd.safeTransfer(msg.sender, amount);
        emit FundsWithdrawn(msg.sender, amount, block.timestamp);
```

```
    }



    function seizeBadDebt(uint amount) external onlyMarginAccount {
        pendingObligation += amount;
        emit BadDebtAccumulated(amount, block.timestamp);
        settlePendingObligation();
    }
```

Recommended Mitigation Steps

Consider making the withdrawals a two step process. The first step requests a withdrawal and marks the time. The second request processes the withdrawal but requires a period of time to elapse since the first step.

To avoid having users constantly having pending withdrawal, each withdrawal should have an expiry time and also a recharge time. The if the second step is not called within expiry amount of time it should be considered invalid. The first step must not be able to be called until recharge time has passed.

Another solution involves a design change where the insurance fund is slowly filled up over time without external deposits. However, this has the disadvantage that bad debts received early in the protocols life time may not have sufficient insurance capital to cover them.

[atvanguard (Hubble) confirmed]

[M-12] AMM Cannot Be `initialize()` Except By Governance

Submitted by kirk-baird

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/AMM.sol#L93-L108
https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/AMM.sol#L730-L734
https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/legos/Governable.sol#L10-L13

The contact `AMM.sol` cannot be initialize unless it is called from the `_governance` address.

This prevents the use of a deployer account and requires the governance to be able to deploy proxy contracts and encode the required arguements. If this is not feasible then the contract cannot be deployed.

Proof of Concept

`initialize()` calls `_setGovernance(_governance);` which will store the governance address.

Following this it will call `syncDeps(_registry);` which has `onlyGovernance` modifier. Thus, if the `msg.sender` of `initialize()` is not the same as the parameter `_governance` then the initialisation will revert.

```
function initialize(
    address _registry,
    address _underlyingAsset,
    string memory _name,
    address _vamm,
    address _governance
) external initializer {
    _setGovernace(_governance);

    vamm = IVAMM(_vamm);
    underlyingAsset = _underlyingAsset;
    name = _name;
    fundingBufferPeriod = 15 minutes;

    syncDeps(_registry);
}
```

Recommended Mitigation Steps

Consider adding the steps manually to `initialize()` . i.e.

```
function initialize(
    address _registry,
    address _underlyingAsset,
```

```
        string memory _name,
        address _vamm,
        address _governance
    ) external initializer {
        _setGovernace(_governance);

        vamm = IVAMM(_vamm);
        underlyingAsset = _underlyingAsset;
        name = _name;
        fundingBufferPeriod = 15 minutes;

        IRegistry registry = IRegistry(_registry);
        clearingHouse = registry.clearingHouse();
        oracle = IOracle(registry.oracle());
}
```

[atvanguard (Hubble) confirmed and resolved]

[M-13] Assets sent from `MarginAccount` to `InsuranceFund` will be locked forever

Submitted by 0xliumin, also found by hyh, minhquanym, and WatchPug

[https://github.com/code-423n4/2022-02-hubble/blob/ed1d885d5dbc2eae24e43c3ecbf291a0f5a52765/contracts/MarginAccount.sol#L377]

Assets sent from MarginAccount to InsuranceFund will be locked forever.

Proof of Concept

The insurance fund doesn't have a way to transfer non-vusd out of the contract.

Assets transferred to the InsuranceFund will be locked forever.

Recommended Mitigation Steps

Have a way for governance to sweep tokens to swap them.

[atvanguard (Hubble) confirmed and commented]:

Yes, this a known issue and already on our roadmap.

moose-code (judge) commented:

The insurance fund contract is also upgradeable so it's a fairly simple fix upgrade and to sweep the tokens out when the time comes - i.e. tokens won't be lost forever. Still would be better to have it in from the start to avoid this process. Considering moving to medium. Assessing other issues first, will circle back.

moose-code (judge) decreased severity to Medium and commented:

Moving to medium as contracts are upgradeable so the tokens can always be collected later. If the contract was non-upgradeable I would have left as high.

[M-14] Liquidation is vulnerable to sandwich attacks

Submitted by danb, also found by leastwood

When an account is liquidated, there is no minimum amount of the swap, which makes it vulnerable for sandwich attacks.

Proof of Concept

Alice's long position can be liquidated, bob notices it and creates a short position, then liquidates her position, thus swapping the base asset to the quote asset, therefore reducing the base asset price,
then he redeems his short position and profits because the price went down.

Recommended Mitigation Steps

Set quoteAssetLimit in `_reducePosition` to prevent the attack.

atvanguard (Hubble) disputed and commented:

This is a known issue and is already documented as a @todo in the code.

moose-code (judge) decreased severity to Medium and commented:

After long discussion we are going to side with warden on this one. The todo is a bit sparse and the warden really digs on what precautions need to be put in place and the ramifications if they are not adhered. In general, think sprinkle of todos should not indemnify issues related around them as this might let things slip through cracks as wardens will ignore these critical pieces.

With the caveat of putting this in the medium and not high risk category.

[M-15] [WP-H7] `InsuranceFund#syncDeps()` may cause users' fund loss

Submitted by WatchPug

https://github.com/code-423n4/2022-02-hubble/blob/ed1d885d5dbc2eae24e43c3ecbf291a0f5a52765/contracts/InsuranceFund.sol#L116-L119

```
function syncDeps(IRegistry _registry) public onlyGovernance {
    vusd = IERC20(_registry.vusd());
    marginAccount = _registry.marginAccount();
}
```

The `Governance` address can call `InsuranceFund.sol#syncDeps()` to change the contract address of `vusd` anytime.

However, since the tx to set a new address for `vusd` can get in between users' txs to deposit and withdraw, in some edge cases, it can result in users' loss of funds.

Proof of Concept

1. Alice deposited `1,000,000 VUSD` to `InsuranceFund`;
2. Gov called `syncDeps()` and set `vusd` to the address of `VUSDv2`;
3. Alice called `withdraw()` with all the `shares` and get back `0 VUSDv2`.

As a result, Alice suffered a fund loss of `1,000,000 VUSD`.

Recommended Mitigation Steps

1. Consider making `vusd` unchangeable;
2. If a possible migration of `vusd` must be considered, consider changing the `syncDeps()` to:

```
function syncDeps(IRegistry _registry) public onlyGovernance {
    uint _balance = balance();
    vusd = IERC20(_registry.vusd());
    require(balance() >= _balance);
    marginAccount = _registry.marginAccount();
}
```

atvanguard (Hubble) acknowledged and commented:

Acknowledging but yes system heavily relies on the admins to do the right thing, the right way. We might remove several such upgradeability rights during a broader refactor of the entire system.

moose-code (judge) decreased severity to Medium and commented:

Downgrading to medium as this is largely admin related.

[M-16] USDC blacklisted accounts can DoS the withdrawal system

Submitted by throttle

DoS of USDC withdrawal system

Proof of Concept

Currently, withdrawals are queued in an array and processed sequentially in a for loop.
However, a `safeTransfer()` to USDC blacklisted user will fail. It will also brick the withdrawal system because the blacklisted user is never cleared.

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/VUSD.sol#L53-L67

Recommended Mitigation Steps

Possible solutions:
1st solution:
Implement 2-step withdrawals:
- In a for loop, increase the user's amount that can be safely withdrawn.
- A user himself withdraws his balance

2nd solution:
Skip blacklisted users in a processWithdrawals loop

atvanguard (Hubble) confirmed

moose-code (judge) commented:

Interesting! Yes, this would be bad.

[M-17] Usage of an incorrect version of Ownable library can potentially malfunction all `onlyOwner` functions

Submitted by robee

The current implementaion is using a non-upgradeable version of the Ownable library. Instead of the upgradeable version: @openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol.

A regular, non-upgradeable Ownable library will make the deployer the default owner in the constructor. Due to a requirement of the proxy-based upgradeability system, no constructors can be used in upgradeable contracts. Therefore, there will be no owner when the contract is deployed as a proxy contract

Recommended Mitigation Steps

Use @openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol and @openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol instead.

And add _*Ownable*init(); at the beginning of the initializer.

Oracle.sol
AMM.sol

Low Risk and Non-Critical Issues

For this contest, 30 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below](https://github.com/code-423n4/2022-02-hubble-findings/issues/10) by **defsec** received the top score from the judge.

The following wardens also submitted reports: [Dravee](), [csanuragjain](), [itsmeSTYJ](), [robee](), [Omik](), [gzeon](), [kenta](), [pauliax](), [ye0lde](), [IllIllI](), [0x1f8b](), [bobi](), [peritoflores](), [leastwood](), [0v3rf10w](), [0xwags](), [Meta0xNull](), [0x0x0x](), [hubble](), [cccz](), [rfa](), [sorrynotsorry](), [danb](), [hyh](), [jayjonah8](), [kirk-baird](), [WatchPug](), [CertoraInc](), *and* [Nikolay]().

[L-01] PREVENT DIV BY 0

On several locations in the code precautions are taken not to divide by 0, because this will revert the code. However on some locations this isn't done.

Oracle price is not checked. That will cause to revert on the several functions.

Proof of Concept

Navigate to the following contract:

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/Oracle.sol#L34

Recommended Mitigation Steps

Recommend making sure division by 0 won't occur by checking the variables beforehand and handling this edge case.

[L-02] Single-step change of governance address is extremely risky

Single-step change of critical governance address and lack of zero address check is extremely risky. If a zero address or incorrect address (private key not available) is used accidentally, or maliciously changed by a compromised governance account

then the entire governance of the protocol is locked forever or lost to an attacker. No governance changes can be made by authorized governance account and protocol will have to be redeployed. The reputation of the protocol will take a huge hit. There may be significant fund lock/loss as well.

Interestingly, this 2-step process is applied to the changing of Strategist address but not Governance address. Governance has more authority in the protocol because it can change the Strategist among other things. So this 2-step should definitely be applied to Governance as well.

Given the magnitude of the impact, i.e. permanent lock of all governance actions, potential lock/loss of funds, and the known/documented failures of wallet opsec, this risk is classified as medium severity.

Proof of Concept

https://github.com/code-423n4/2022-02-hubble/blob/main/contracts/legos/Governable.sol#L20

Recommended Mitigation Steps

Change of the most critical protocol address i.e. governance should be timelocked and be a 2-step process: approve+claim in two different transactions, instead of a single-step change.

[L-03] Front-runnable Initializers

All contract **initializers** were missing access controls, allowing any user to initialize the contract. By front-running the contract deployers to initialize the contract, the incorrect parameters may be supplied, leaving the contract needing to be redeployed.

Proof of Concept

1. Navigate to the following contracts:

    - https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/AMM.sol#L93

    - https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/Oracle.sol#L20
    - https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/VUSD.sol#L38
    - https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccount.sol#L121
2. Initialize functions does not have access control. They are vulnerable to front-running.

Recommended Mitigation Steps

While the code that can be run in contract constructors is limited, setting the owner in the contract's constructor to the `msg.sender` and adding the `onlyOwner` modifier to all **initializers** would be a sufficient level of access control.

[L-04] Incompatibility With Rebasing/Deflationary/Inflationary tokens

The protocol do not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest.

Proof of Concept

Navigate to the following contracts:

- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccount.sol#L155
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccountHelper.sol#L29

Recommended Mitigation Steps

- Ensure that to check previous balance/after balance equals to amount for any rebasing/inflation/deflation
- Add support in contracts for such tokens before accepting user-supplied tokens
- Consider supporting deflationary / rebasing / etc tokens by extra checking the balances before/after or strictly inform your users not to use such tokens if they don't want to lose them.

[L-05] Missing zero-address check in constructors and the setter functions

Missing checks for zero-addresses may lead to infunctional protocol, if the variable addresses are updated incorrectly.

Proof of Concept

Navigate to the following contract functions:

- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccountHelper.sol#L19
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/VUSD.sol#L39
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/legos/Governable.sol#L16
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/InsuranceFund.sol#L35
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccount.sol#L121

Recommended Mitigation Steps

Consider adding zero-address checks in the discussed constructors:
require(newAddr != address(0));.

[L-06] Missing events for governor only functions that change critical parameters

The governor only functions that change critical parameters should emit events. Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. The alternative of directly querying on-chain contract state for such changes is not considered practical for most users/usages.

Missing events and timelocks do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in liquidity which could negatively impact protocol TVL and reputation.

There are owner functions that do not emit any events in the contracts.

Proof of Concept

Navigate to the following contracts:

- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccount.sol#L616
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/legos/Governable.sol#L19
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/Oracle.sol#L162
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/VUSD.sol#L74
- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/AMM.sol#L722

- https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/AMM.sol#L737

See similar High-severity H03 finding OpenZeppelin's Audit of Audius and Medium-severity M01 finding OpenZeppelin's Audit of UMA Phase 4.

Recommended Mitigation Steps

Add events to all admin/privileged functions that change critical parameters.

[L-07] Deprecated safeApprove() function

Detailed description of the impact of this finding.

Using this deprecated function can lead to unintended reverts and potentially the locking of funds. A deeper discussion on the deprecation of this function is in OZ issue #2219. The OpenZeppelin ERC20 `safeApprove()` function has been deprecated, as seen in the comments of the OpenZeppelin code.

Proof of Concept

Navigate to the following contract functions:

https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccountHelper.sol#L24

Recommended Mitigation Steps

As suggested by the OpenZeppelin comment, replace `safeApprove()` with `safeIncreaseAllowance()` or `safeDecreaseAllowance()` instead.

[L-08] The Contract Should Approve(0) first

Some tokens (like USDT L199) do not work when changing the allowance from an existing non-zero allowance value.
They must first be approved by zero and then the actual allowance must be approved.

```
IERC20(token).approve(address(operator), 0);
```

```
    IERC20(token).approve(address(operator), amount);
```

Proof of Concept

Navigate to the following contract functions:

https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/MarginAccountHelper.sol#L24

Recommended Mitigation Steps

Approve with a zero amount first before setting the actual amount.

[L-09] Missing Pause Modifier On the InsuranceFunds contract

In case a hack occurs or an exploit is discovered, the team should be able to pause functionality until the necessary changes are made to the system. The deposits should be paused with Pause modifier.

Proof of Concept

Navigate to the following contract functions:

https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/InsuranceFund.sol#L39

There is no pause mechanism has been defined.

Recommended Mitigation Steps

Pause functionality on the contract would have helped secure the funds quickly.

[N-01] Use of Block.timestamp

Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion for further details), locking funds for periods of time, and various state-changing conditional statements that are time-dependent. Miners have the ability to adjust timestamps slightly, which can prove to be dangerous if block timestamps are used incorrectly in smart contracts.

Proof of Concept

Navigate to the following contract:

https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/legos/HubbleBase.sol#L49

Recommended Mitigation Steps

Block timestamps should not be used for entropy or generating random numbers—i.e., they should not be the deciding factor (either directly or through some derivation) for winning a game or changing an important state.

Time-sensitive logic is sometimes required; e.g., for unlocking contracts (time-locking), completing an ICO after a few weeks, or enforcing expiry dates. It is sometimes recommended to use block.number and an average block time to estimate times; with a 10 second block time, 1 week equates to approximately, 60480 blocks. Thus, specifying a block number at which to change a contract state can be more secure, as miners are unable to easily manipulate the block number.

[N-02] Missing Re-entrancy Guard

The re-entrancy guard is missing on the Eth anchor interaction. The external router interaction can cause to the re-entrancy vulnerability.

Proof of Concept

Navigate to the following contract functions:

https://github.com/code-423n4/2022-02-hubble/blob/8c157f519bc32e552f8cc832ecc75dc381faa91e/contracts/InsuranceFund.sol#L39

Recommended Mitigation Steps

Follow the check effect interaction pattern or put re-entrancy guard.

[atvanguard (Hubble) commented](#):

Good QA report.

[moose-code (judge) commented](#):

Lots of good insights here 💯

Gas Optimizations

For this contest, 21 reports were submitted by wardens detailing gas optimizations. The [report highlighted below](#) by **Dravee** received the top score from the judge.

The following wardens also submitted reports: [IllIllI](#), [Jujic](#), [WatchPug](#), [csanuragjain](#), [defsec](#), [rfa](#), [robee](#), [throttle](#), [0v3rf10w](#), [CertoraInc](#), [d4rk](#), [gzeon](#), [kenta](#), [sorrynotsorry](#), [0x1f8b](#), [Meta0xNull](#), [Omik](#), [Tomio](#), [pauliax](#), *and* [danb](#).

Table of Contents

See [original submission](#) for table of contents.

Foreword

- **Storage-reading optimizations**

The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). In the paragraphs below, please see the `@audit-issue` tags in the pieces of code's comments for more information about SLOADs that could be saved by caching the mentioned **storage** variables in **memory** variables.

- **Unchecking arithmetics operations that can't underflow/overflow**

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation, or the operation doesn't depend on user input), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

- `@audit` **tags**

The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.

Summary

- One pattern that was often seen is caching structs in memory when it's not needed. A copy in memory of a storage struct will trigger as many SLOADs as there are slots. If the struct's fields are only read once, or if the number of storage reading would be inferior to the number of slots: don't cache the struct in memory.

File: AMM.sol

function initialize()

```
093:     function initialize(
094:         address _registry,
095:         address _underlyingAsset,
096:         string memory _name,//@audit readonly: calldata
097:         address _vamm,
098:         address _governance
099:     ) external initializer {
100:         _setGovernace(_governance);
101: 
102:         vamm = IVAMM(_vamm);
103:         underlyingAsset = _underlyingAsset;
104:         name = _name;
105:         fundingBufferPeriod = 15 minutes;
106: 
107:         syncDeps(_registry);
108:     }
```

Use `calldata` instead of `memory` for `string _name`

An external function passing a readonly variable should mark it as `calldata` and not `memory`

function openPosition()

```
113:     function openPosition(address trader, int256 baseAssetQuanti
```

```
114:         override
115:         external
116:         onlyClearingHouse
117:         returns (int realizedPnl, uint quoteAsset, bool isPositi
118:     {
119:         require(ammState == AMMState.Active, "AMM.openPosition.n
120:         Position memory position = positions[trader]; //@audit 3
121:         bool isNewPosition = position.size == 0 ? true : false;
122:         Side side = baseAssetQuantity > 0 ? Side.LONG : Side.SHO
123:         if (isNewPosition || (position.size > 0 ? Side.LONG : Si
124:             // realizedPnl = 0;
125:             quoteAsset = _increasePosition(trader, baseAssetQuan
126:             isPositionIncreased = true;
127:         } else {
128:             (realizedPnl, quoteAsset, isPositionIncreased) = _op
129:         }
130:         _emitPositionChanged(trader, realizedPnl);
131:     }
```

Do not cache `positions[trader]` in memory

As a copy in memory of a struct makes as many SLOADs as there are slots, here a copy costs 3 SLOADs:

```
41:     struct Position {
42:         int256 size;
43:         uint256 openNotional;
44:         int256 lastPremiumFraction;
45:     }
```

However, only the `size` field is read twice. Therefore, only this field should get cached: `int256 _size = positions[trader].size;`

function liquidatePosition()

```
133:     function liquidatePosition(address trader)
134:         override
135:         external
136:         onlyClearingHouse
```

```
137:         returns (int realizedPnl, uint quoteAsset)
138:     {
139:         // don't need an ammState check because there should be r
140:         Position memory position = positions[trader]; //@audit 3
141:         bool isLongPosition = position.size > 0 ? true : false;
142:         // sending market orders can fk the trader. @todo put sor
143:         if (isLongPosition) {
144:             (realizedPnl, quoteAsset) = _reducePosition(trader,
145:         } else {
146:             (realizedPnl, quoteAsset) = _reducePosition(trader,
147:         }
148:         _emitPositionChanged(trader, realizedPnl);
149:     }
```

Do not cache `positions[trader]` in memory

Similar to Do not cache `positions[trader]` in memory.

However, only the size field is read 3 times. Therefore, only this field should get cached: `int256 _size = positions[trader].size;`

function removeLiquidity()

```
133:     function liquidatePosition(address trader)
134:         override
135:         external
136:         onlyClearingHouse
137:         returns (int realizedPnl, uint quoteAsset)
138:     {
139:         // don't need an ammState check because there should be r
140:         Position memory position = positions[trader]; //@audit 3
141:         bool isLongPosition = position.size > 0 ? true : false;
142:         // sending market orders can fk the trader. @todo put sor
143:         if (isLongPosition) {
144:             (realizedPnl, quoteAsset) = _reducePosition(trader,
145:         } else {
146:             (realizedPnl, quoteAsset) = _reducePosition(trader,
147:         }
148:         _emitPositionChanged(trader, realizedPnl);
149:     }
```

Do not cache `positions[maker]` in memory

Similar to [Do not cache `positions[trader]` in memory](). However, here, even the fields shouldn't get cached, as they are read only once:

```
220:         Position memory _taker = positions[maker];//@audit 3 SLO
...
233:             _taker.size,
234:             _taker.openNotional
```

Therefore, use `220: Position storage _taker = positions[maker];`

Do not cache `makers[maker]` in memory

Similarly, a copy in memory for `Maker` costs 7 SLOADs:

```
48:     struct Maker {
49:         uint vUSD;
50:         uint vAsset;
51:         uint dToken;
52:         int pos; // position
53:         int posAccumulator; // value of global.posAccumulator unt
54:         int lastPremiumFraction;
55:         int lastPremiumPerDtoken;
56:     }
```

Here, caching the first 5 fields in memory is enough.

function getNotionalPositionAndUnrealizedPnl()

```
395:     function getNotionalPositionAndUnrealizedPnl(address trader)
396:         override
397:         external
398:         view
399:         returns(uint256 notionalPosition, int256 unrealizedPnl, 
400:     {
401:         Position memory _taker = positions[trader];//@audit 3 SL
402:         Maker memory _maker = makers[trader];//@audit 7 SLOADs v
```

```
403:
404:         (notionalPosition, size, unrealizedPnl, openNotional) = \
405:             _maker.dToken,
406:             _maker.vUSD,
407:             _maker.vAsset,
408:             _taker.size,
409:             _taker.openNotional
410:         );
411:     }
```

Do not cache `positions[trader]` in memory

Here, we need `Position storage _taker = positions[trader];`

Do not cache `makers[trader]` in memory

Here, we need `Maker storage _maker = makers[trader];`

function getPendingFundingPayment()

```
425:         Position memory taker = positions[trader];//@audit 3 SLO/
...
434:         Maker memory maker = makers[trader];//@audit 7 SLOADs vs
```

Do not cache `positions[trader]` in memory

Here, we need `Position storage _taker = positions[trader];`

Do not cache `makers[trader]` in memory

Here, we need `Maker storage _maker = makers[trader];`

function getTakerNotionalPositionAndUnrealizedPnl()

```
458:     function getTakerNotionalPositionAndUnrealizedPnl(address tra
459:         Position memory position = positions[trader];//@audit 3 S
460:         if (position.size > 0) {
461:             takerNotionalPosition = vamm.get_dy(1, 0, position.s
462:             unrealizedPnl = takerNotionalPosition.toInt256() - p
463:         } else if (position.size < 0) {
```

```
464:                takerNotionalPosition = vamm.get_dx(0, 1, (-position
465:                unrealizedPnl = position.openNotional.toInt256() - ta
466:            }
467:        }
```

Do not cache `positions[trader]` in memory

Here, we need to cache these fields: `size` and `openNotional`

function _emitPositionChanged()

```
527:        function _emitPositionChanged(address trader, int256 realize
528:            Position memory position = positions[trader];//@audit 3 S
529:            emit PositionChanged(trader, position.size, position.oper
530:        }
```

Do not cache `positions[trader]` in memory

Here, we need `Position storage _taker = positions[trader];`

function _openReversePosition()

```
584:        function _openReversePosition(address trader, int256 baseAsse
585:            internal
586:            returns (int realizedPnl, uint quoteAsset, bool isPositio
587:        {
588:            Position memory position = positions[trader];//@audit 3 S
589:            if (abs(position.size) >= abs(baseAssetQuantity)) {
590:                (realizedPnl, quoteAsset) = _reducePosition(trader, b
591:            } else {
592:                uint closedRatio = (quoteAssetLimit * abs(position.si
593:                (realizedPnl, quoteAsset) = _reducePosition(trader,
594:
595:                // this is required because the user might pass a ve
596:                if (quoteAssetLimit >= quoteAsset) {
597:                    quoteAssetLimit -= quoteAsset; //@audit uncheck
598:                }
599:                quoteAsset += _increasePosition(trader, baseAssetQuan
600:                isPositionIncreased = true;
601:            }
```

```
602:    }
```

Do not cache `positions[trader]` in memory

Here, we need to cache the `size` field

Unchecked block L597

This line can't underflow due to the condition L596. Therefore, it should be wrapped in an `unchecked` block

function _calcTwap()

```
656:    function _calcTwap(uint256 _intervalInSeconds)
657:        internal
658:        view
659:        returns (uint256)
660:    {
661:        uint256 snapshotIndex = reserveSnapshots.length - 1; //@a
662:        uint256 currentPrice = reserveSnapshots[snapshotIndex].la
663:        if (_intervalInSeconds == 0) {
664:            return currentPrice;
665:        }
666:
667:        uint256 baseTimestamp = _blockTimestamp() - _intervalInS
668:        ReserveSnapshot memory currentSnapshot = reserveSnapshot
669:        // return the latest snapshot price directly
670:        // if only one snapshot or the timestamp of latest snaps
671:        if (reserveSnapshots.length == 1 || currentSnapshot.time
...
675:        uint256 previousTimestamp = currentSnapshot.timestamp;
676:        uint256 period = _blockTimestamp() - previousTimestamp;
677:        uint256 weightedPrice = currentPrice * period;
678:        while (true) {
...
680:            if (snapshotIndex == 0) {
681:                return weightedPrice / period;
682:            }
...
684:            snapshotIndex = snapshotIndex - 1; //@audit uncheck
685:            currentSnapshot = reserveSnapshots[snapshotIndex];
686:            currentPrice = reserveSnapshots[snapshotIndex].lastP
```

```
...
689:                if (currentSnapshot.timestamp <= baseTimestamp) {
...
698:                uint256 timeFraction = previousTimestamp - currentSna
...
701:                previousTimestamp = currentSnapshot.timestamp;
```

Do not cache `reserveSnapshots[snapshotIndex]` in memory

Here, we need to cache the `timestamp` field. Copying the struct in memory costs 3 SLOADs.

Cache `reserveSnapshots.length` in memory

This would save 1 SLOAD

Unchecked block L684

This line can't underflow due to the condition L680-L682. Therefore, it should be wrapped in an `unchecked` block

Use the cache for calculation

As we already have `currentSnapshot = reserveSnapshots[snapshotIndex];` : use it here: `currentPrice = currentSnapshot.lastPrice;`

File: ClearingHouse.sol

function _disperseLiquidationFee()

```
210:     function _disperseLiquidationFee(uint liquidationFee) interna
211:         if (liquidationFee > 0) {
212:             uint toInsurance = liquidationFee / 2;
213:             marginAccount.transferOutVusd(address(insuranceFund)
214:             marginAccount.transferOutVusd(_msgSender(), liquidat
215:         }
216:     }
```

Unchecked block L214

This line can't underflow due to the condition L212. Therefore, it should be wrapped in an `unchecked` block

File: InsuranceFund.sol

function pricePerShare()

```
File: InsuranceFund.sol
094:     function pricePerShare() external view returns (uint) {
095:         uint _totalSupply = totalSupply();
096:         uint _balance = balance();
097:         _balance -= Math.min(_balance, pendingObligation); //@au
098:         if (_totalSupply == 0 || _balance == 0)
099:             return PRECISION;
100:         }
101:         return _balance * PRECISION /
_totalSupply;
102:     }
```

Unchecked block L97

This line can't underflow for obvious mathematical reasons (`_balance` substracting at most itself). Therefore, it should be wrapped in an `unchecked` block

File: Oracle.sol

function getUnderlyingTwapPrice()

Unchecked block L81

This line can't underflow due to L76-L79. Therefore, it should be wrapped in an `unchecked` block

File: Interfaces.sol

struct Collateral

Tight packing structs to save slots

While this file is out of scope, it deeply impacts MarginAccount.sol. I suggest going from:

```
94:     struct Collateral {
95:         IERC20 token; //@audit 20 bytes
96:         uint weight; //@audit 32 bytes
97:         uint8 decimals; //@audit 1 byte
98:     }
```

to

```
94:     struct Collateral {
95:         IERC20 token; //@audit 20 bytes
96:         uint8 decimals; //@audit 1 byte
97:         uint weight; //@audit 32 bytes
98:     }
```

To save 1 slot per array element in MarginAccount.sol's storage

File: MarginAccount.sol

function _getLiquidationInfo()

```
460:     function _getLiquidationInfo(address trader, uint idx) intern
461:         require(idx > VUSD_IDX && idx < supportedCollateral.leng
462:         (buffer.status, buffer.repayAble, buffer.incentivePerDol
463:         if (buffer.status == IMarginAccount.LiquidationStatus.IS_
464:             Collateral memory coll = supportedCollateral[idx];//(
465:             buffer.priceCollateral = oracle.getUnderlyingPrice(a
466:             buffer.decimals = coll.decimals;
467:         }
468:     }
```

Do not cache `supportedCollateral[idx]` in memory

Here, we need `Collateral storage coll = supportedCollateral[idx];` . Copying the struct in memory costs 3 SLOADs.

function _transferOutVusd()

Unchecked block L588

This line can't underflow due to L583. Therefore, it should be wrapped in an `unchecked` block

File: VUSD.sol

function processWithdrawals()

```
    53:     function processWithdrawals() external {
    54:         uint reserve = reserveToken.balanceOf(address(this));
    55:         require(reserve >= withdrawals[start].amount, 'Cannot pro
    56:         uint i = start;//@audit start SLOAD 2
    57:         while (i < withdrawals.length && (i - start) <= maxWithdr
    58:             Withdrawal memory withdrawal = withdrawals[i]; //@aud
    59:             if (reserve < withdrawal.amount) {
    60:                 break;
    61:             }
    62:             reserveToken.safeTransfer(withdrawal.usr, withdrawal.
    63:             reserve -= withdrawal.amount;  //@audit uncheck (see
    64:             i += 1;
    65:         }
    66:         start = i;
    67:     }
```

Unchecked block L57-L65

The whole while-loop can't underflow. Therefore, it should be wrapped in an `unchecked` block

Cache `start` in memory

Cache `start` in memory as `initialStart` and use it L55 + L57 (compare `i` to it in the while-loop)

General Recommendations

Variables

No need to explicitly initialize variables with default values

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default

value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Instances include:

```
ClearingHouse.sol:122:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:130:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:170:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:194:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:251:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:263:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:277:        for (uint i = 0; i < amms.length; i++) 
InsuranceFund.sol:52:         uint shares = 0;
MarginAccount.sol:31:    uint constant VUSD_IDX = 0;
MarginAccount.sol:331:        for (uint i = 0; i < idxs.length; i++) 
MarginAccount.sol:521:        for (uint i = 0; i < assets.length; i++
MarginAccount.sol:552:        for (uint i = 0; i < _collaterals.lengt
MarginAccountHelper.sol:13:    uint constant VUSD_IDX = 0;
```

I suggest removing explicit initializations for default values.

Pre-increments cost less gas compared to post-increments

Comparisons

`> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

`> 0` in require statements are used in the following location(s):

```
AMM.sol:487:        require(baseAssetQuantity > 0, "VAMM._long: baseA
AMM.sol:511:        require(baseAssetQuantity < 0, "VAMM._short: base
ClearingHouse.sol:51:        require(_maintenanceMargin > 0, "_mainte
MarginAccount.sol:150:        require(amount > 0, "Add non-zero margi
Oracle.sol:153:        require(_round > 0, "Not enough history");
```

I suggest you change `> 0` with `!= 0` in require statements. Also, enable the Optimizer.

For-Loops

An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
ClearingHouse.sol:122:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:130:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:170:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:194:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:251:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:263:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:277:        for (uint i = 0; i < amms.length; i++)
MarginAccount.sol:331:        for (uint i = 0; i < idxs.length; i++)
MarginAccount.sol:373:        for (uint i = 1 /* skip vusd */; i < as
MarginAccount.sol:521:        for (uint i = 0; i < assets.length; i++
MarginAccount.sol:552:        for (uint i = 0; i < _collaterals.lengt
```

`++i` costs less gas compared to `i++`

`++i` costs less gas compared to `i++` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration)

`i++` increments `i` and returns the initial value of `i`. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But `++i` returns the actual incremented value:

```
uint i = 1;
++i; // == 2 and i == 2 too, so no need for a temporary variable
```

In the first case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`

Instances include:

```
ClearingHouse.sol:122:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:130:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:170:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:194:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:251:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:263:        for (uint i = 0; i < amms.length; i++)
ClearingHouse.sol:277:        for (uint i = 0; i < amms.length; i++)
MarginAccount.sol:331:        for (uint i = 0; i < idxs.length; i++)
MarginAccount.sol:373:        for (uint i = 1 /* skip vusd */; i < as
MarginAccount.sol:521:        for (uint i = 0; i < assets.length; i++
MarginAccount.sol:552:        for (uint i = 0; i < _collaterals.lengt
```

I suggest using `++i` instead of `i++` to increment the value of an uint variable.

Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Instances include:

```
ClearingHouse.sol:122:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:130:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:170:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:194:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:251:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:263:        for (uint i = 0; i < amms.length; i++) 
ClearingHouse.sol:277:        for (uint i = 0; i < amms.length; i++) 
MarginAccount.sol:331:        for (uint i = 0; i < idxs.length; i++) 
MarginAccount.sol:373:        for (uint i = 1 /* skip vusd */; i < as
MarginAccount.sol:521:        for (uint i = 0; i < assets.length; i++
MarginAccount.sol:552:        for (uint i = 0; i < _collaterals.lengt
```

The code would go from:

```
for (uint256 i; i < numIterations; i++) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

The risk of overflow is inexistant for a `uint256` here.

Arithmetics

Shift Right instead of Dividing by 2

A division by 2 can be calculated by shifting one to the right.

While the `DIV` opcode uses 5 gas, the `SHR` opcode only uses 3 gas. Furthermore, Solidity’s division operation also includes a division-by-0 prevention which is

bypassed using shifting.

I suggest replacing `/ 2` with `>> 1` here:

```
ClearingHouse.sol:212:            uint toInsurance = liquidationFee /
```

Errors

Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

Revert strings > 32 bytes are here:

```
AMM.sol:487:        require(baseAssetQuantity > 0, "VAMM._long: baseA
AMM.sol:511:        require(baseAssetQuantity < 0, "VAMM._short: baseA
ClearingHouse.sol:84:             require(isAboveMinAllowableMargin(tra
ClearingHouse.sol:101:        require(isAboveMinAllowableMargin(maker
MarginAccount.sol:174:        require(margin[VUSD_IDX][trader] >= 0,
MarginAccount.sol:354:        require(notionalPosition == 0, "Liquida
MarginAccount.sol:453:        require(repay <= maxRepay, "Need to repa
```

I suggest shortening the revert strings to fit in 32 bytes, or that using custom errors as described next.

Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

See original submission for instances.

I suggest replacing revert strings with custom errors.

atvanguard (Hubble) confirmed and commented:

Amazing report! ⭐

moose-code (judge) commented:

Really detailed and well constructed report. 💯

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.