Hubble

Smart Contract Security Assessment

12.04.2021

ABSTRACT

Dedaub was commissioned to audit the Hubble protocol on Avalanche. The Hubble protocol is a fully decentralized implementation of a perpetual futures exchange. Unlike Futures contracts in TradFi, Perpetual futures don't have an expiration and rely on continuous funding payments to maintain their economic viability. Perpetual futures have been popularized by the FTX exchange and offer highly leveraged positions of underlying cryptocurrencies against a quote token (typically USD). This audit report covers commit hash `9da65fd659ea74b3a8e0dacda2afa102d1808b6a`. Two auditors and a mathematician worked over the codebase over three weeks.

The codebase appears to be well-tested and covers many corner cases. Documentation was provided, but not in a lot of detail. The team auditing this protocol was already knowledgeable in vAMM-based perpetual future exchanges and could second-guess the intent of the protocol design from its code most of the time. The audit did not include Curve Math, HubbleViewer.sol, standard libraries, or known vulnerabilities, listed in another section.

Hubble, which enables decentralized perpetual futures, offers users the ability to gain highly leveraged *price exposure* to synthetic versions of crypto-assets. The underlying protocol implements a complex accounting model, and allows users to trade or stake virtual tokens, via multiple layers of indirection (namely: `ClearingHouse.sol` and `AMM.sol`), through a modified Curve Crypto Pools AMMs initialized with these VTokens (synthetic assets).

Hubble allows users to hold various positions on one address for multiple virtual base tokens.

Security Opinion

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") was a secondary consideration, however intensive efforts were made to check the correct application of the mathematical formulae in the reviewed code. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. Although a number of simulations have been carried out, the crypto-economic effectiveness of this specific design is not certain. Therefore, the financial viability of this protocol in real market conditions cannot be fully established.

In terms of architecture, Dedaub notes that there are several design decisions that ensure the economic security of the protocol. These include:

Dual mode margin calculation: Hubble uses curve crypto's internal TWAP to compare the vAMM's internal price, and compares this to the index price derived from an Oracle. When there's a significant deviation between the two, the margin is derived by combining both in a clever way. If the user is opening or increasing a position (maker or taker), the margin is calculated by taking the worst margin fraction out of both oracle and vAMM calculations. This would impede users from opening large positions in such a state. On the other hand, when a user wants to liquidate a position, the best margin fraction for that user (using the Oracle and internal vAMM price) is calculated. This means that some positions will not be able to be increased while at the same time cannot be liquidated either.

Dedaub also notes that there are architectural decisions, and limitations of decentralization, which could potentially threaten the crypto-economic security of Hubble:

- Dual mode margin calculation protects takers but does not offer sufficient protection against maker liquidation.
- Unbounded slippage is possible when when opening / closing positions, or when liquidating positions

In terms of protocol completion, the protocol appears to be in the final stage of its development. A few contracts are lacking a complete implementation with certain checks or functionality still under development. This means that further development is needed, which could render the current security assessment partially insufficient.

Next we list identified issues that the protocol team was already aware of at the beginning of the audit and is already actively working on (or is going to work on before mainnet launch):

1. Assets that are sent from the MarginAccount contract to the InsuranceFund contract when a liquidation happens will stay locked forever, as the functionality to manage/trade these assets has not been implemented yet. This can also lead to the InsuranceFund not being able to cover its obligations, effectively pausing deposits and withdrawals.
2. Users are able to front-run bad debt settlements and withdraw their InsuranceFund share to avoid insurance costs.
3. The Oracle implementation is missing several important checks to be considered robust.
4. Partial liquidation and slippage checks need to be implemented in `AMM::forceRemoveLiquidity`.
5. Safeguards around the price of liquidation need to be implemented in `AMM::liquidatePosition`.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Force Liquidations of Makers	**DISMISSED**

The crypto-economic design of this protocol can lead to force-liquidation of Makers through very small price movements. The following design elements make it easy to force liquidate makers:

- Curve-Crypto AMM can yield the same price with different pool compositions
- Spread limit is hard to trigger with single transactions

Scenario: Bob wants to force liquidate Alice's maker position to perform a liquidation slippage sandwich. *[Note: the following figures are approximate]*

1. With a small amount of margin, Alice opens a maker position: $3000 + 0.5ETH, when ETH is at $2000. Note that the pool is not perfectly balanced.
2. Bob opens a large short position, say 10ETH, moving ETH price to $1900.
3. The pool's composition changed significantly with one swap, but not the price.
4. Alice's position is now around $1100 + 1.5ETH, so `openNotional` = 1900 and `position` = 1
5. Alice's *maker debt* is $6000
6. Alice's `notionalPosition` is $7900

The result is that with < 5% price change, Alice's margin fraction has decreased by 25%

M2	Unbonding process is easily circumvented	**DISMISSED**

The unbonding process can be easily circumvented through a variation on the Sybil attack. Unbonding liquidity at will enables other attacks such as liquidity frontrunning.

Scenario: Alice wants to add x amount of liquidity, and be able to withdraw $x/3$ of her liquidity on any one day. We assume that the withdrawal period is N days and the unbonding period is M days. This means that using the following strategy, alice can always remove Nx/M liquidity, like so:

1. Alice deposits Nx/M each day for M days on M different addresses
2. After M days, Alice goes through each address where the withdrawal expired and requests unbonding again.
3. At any day, after the first M days, alice can withdraw up to Nx/M of her liquidity.

M3	Unbonding amount is not reset on liquidation	**RESOLVED**

A maker's liquidation calls method `AMM::forceRemoveLiquidity`, which in turn calls `AMM::_removeLiquidity` and operates in the same manner as the regular `removeLiquidity` thereafter, but does not reset a pending unbonding amount that the maker might have.

The function `AMM::removeLiquidity` on the other hand, deducts the unbonding amount accordingly:

```
Maker storage _maker = _makers[maker];
_maker.unbondAmount -= amount;
```

M4	Cascading Liquidations	**ACKNOWLEDGED**

The risk of cascading liquidations in Hubble are relatively high, especially where maker liquidations are concerned. Takers are relatively protected from triggering liquidations of other takers due to the dual mode margin fraction mechanism (which uses oracle prices in cases of large divergences between mark and index prices). However, a taker liquidation can trigger a maker liquidation (see M1). In turn the removal of maker liquidity makes the price derived via `Swap::get_dy` and `Swap::get_dx` lower.

The following are our inferred cascading liquidation risks:

- Taker liquidation triggering a taker liquidation (low)
- Maker liquidation triggering a taker liquidation (medium, effect of swap price movement in addition to the effect of removal of liquidity)
- Maker liquidation triggering a maker liquidation (high, see M1)
- Taker liquidation triggering a maker liquidation (high, see M1)

M5	Insurance stakers who double as liquidators can increase their share of the pool	**RESOLVED**

[This issue was partially known to the developers]

If an insurance staker also doubles as a liquidator, then they can:

1. Withdraw their insurance contribution
2. Liquidate bad debt
3. Settle bad debt using other user's insurance stake
4. Re-deposit their stake again

The liquidator/staker now owns a larger portion of the pool. This effect can be compounded.

M6	Opening multiple tiny positions to make liquidations unprofitable	**RESOLVED**

There are no restrictions on the minimum size of the position a user can open and on the minimum amount of collateral he should deposit when an account is opened. A really small position will be unprofitable for an arbitrageur to liquidate.

An adversary could take advantage of this fact and open a huge number of tiny positions, using different accounts. The adversary might not be able to get a direct profit from such an approach, but since these positions are going to stay open for a long time, as no one will have a profit by liquidating them, they can significantly shift the price of the vAMM with small risk. To safeguard against such attacks we suggest that a lower bound on the position size and collateral should be used.

M7	Liquidating own tiny maker position to profit from the fixed fee	**RESOLVED**

As discussed in issue M6, one can open a however small position they want. The same is true when providing liquidity. On the other hand the incentive fee for liquidating a maker, i.e., someone that provides liquidity, is fixed and it's 20 dollars as defined in `ClearingHouse::fixedMakerLiquidationFee`. Thus, one could provide really tiny amounts of liquidity (with tiny amounts of collateral backing it) and liquidate themselves with another account to make a profit from the liquidation fee. Networks with small transaction fees (e.g., Avalanche) or L2 solutions could make such an attack really profitable, especially if executed on a large scale.

M8	ClearingHouse::isMaker does not take into account maker's ignition share	**RESOLVED**

Method ClearingHouse::isMaker checks if a user is a maker by implementing the following check:

```
function isMaker(address trader) override public view returns(bool) {
    uint numAmms = amms.length;
    for (uint i; i < numAmms; ++i) {
        IAMM.Maker memory maker = amms[i].makers(trader);
        if (maker.dToken > 0) {
```

```
            return true;
        }
    }
    return false;
}
```

However, the AMM could still be in the ignition phase, meaning that the maker could have provided liquidity that is not yet reflected in `maker.dToken` but in `maker.ignition`. This omission could allow liquidation of a user's taker positions before its maker positions, which is something undesirable, as defined by the `liquidate` and `liquidateTaker` methods of ClearingHouse.

M9	Liquidation Slippage Sandwich Attack	**ACKNOWLEDGED**

[The attack is related to already known issues, but is documented in more detail here]

1. Alice has a long position that is underwater
2. Bob opens a large short position
3. Bob liquidates Alice. This triggers a swap in the same direction as Bob's position and causes slippage.
4. Bob closes his position, and profits on the slippage at the expense of Alice.

M10	Self close bad debt attack	**DISMISSED**

This is a non-specific attack on the economics of the protocol.

1. Alice opens a short position using account A
2. Alice opens a large long position using account B
3. In the meantime, the market moves up.
4. Alice closes her under-collateralized position A. Bad debt occurs.
5. Alice can now close position B and realize her profit

LOW SEVERITY:

ID	Description	STATUS
L1	`removeLiquidity` followed by `addLiquidity` is not position neutral	**ACKNOWLEDGED**
Maker debt, calculated as the vUSD amount * 2 when the liquidity was added never changes. If the maker has gained out of her impermanent position, e.g., through fees, this is not accounted for, in certain kinds of liquidations (via oracle). However, if the maker now removes their liquidity, closes their impermanent position and adds the same amount of liquidity, the debt is reset to a different amount.		
L2	`ClearingHouse blacklisting checks are incomplete`	**RESOLVED**
The `ClearingHouse` contract can set and use a `Blacklist` contract to ban certain users from opening new positions. However, these same users are not blacklisted from providing liquidity to the protocol, i.e., having impermanent positions, which can be turned into permanent ones when the liquidity is removed. Although this form of opening positions is not controllable, it would be better if blacklisted users were also banned from providing liquidity.		
L3	`VUSD::processWithdrawals could potentially be reentered`	**RESOLVED**
`VUSD::processWithdrawals` of the VUSD contract calls method `safeTransfer` on the `reserveToken` defined in VUSD.		

```
function processWithdrawals() external whenNotPaused {
  uint reserve = reserveToken.balanceOf(address(this));
  require(reserve >= withdrawals[start].amount, 'Cannot process
    withdrawals at this time: Not enough balance');
  uint i = start;
  while (i < withdrawals.length && (i - start) < maxWithdrawalProcesses) {
    Withdrawal memory withdrawal = withdrawals[i];
    if (reserve < withdrawal.amount) {
      break;
    }
    reserve -= withdrawal.amount;
    reserveToken.safeTransfer(withdrawal.usr, withdrawal.amount);
    i += 1;
  }
  start = i;
}
```

In the unlikely scenario that the `safeTransfer` method (or a method `safeTransfer` calls internally) of `reserveToken` allows calling an arbitrary contract, then that contract can reenter the `processWithdrawals` method. As the `start` storage variable will not have been updated (it is updated at the very end of the method), the same withdrawal will be executed twice if the contract's `reserveToken` balance is sufficient. Actually, if reentrancy is possible, the whole balance of the contract can be drained by reentering multiple times. It is easier to perform this attack if the attacker's withdrawal is the first to be executed, which is actually not hard to achieve. This vulnerability is highly unlikely, as it requires the execution reaching an untrusted contract, still we suggest adding a reentrancy guard (minor overhead) to completely remove the possibility of such a scenario.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

A1	Gas usage increases quadratically to positions	**ACKNOWLEDGED**

Whenever a user's position is modified, maintained or liquidated, all of the user's token positions need to be queried (both maker and taker). For instance, this happens in `ClearingHouse::getTotalNotionalPositionAndUnrealizedPnl`

```
for (uint i; i < numAmms; ++i) {
  if (amms[i].isOverSpreadLimit()) {
    (_notionalPosition, _unrealizedPnl) =
      amms[i].getOracleBasedPnl(trader, margin, mode);
  } else {
    (_notionalPosition, _unrealizedPnl,,) =
      amms[i].getNotionalPositionAndUnrealizedPnl(trader);
  }
  notionalPosition += _notionalPosition;
  unrealizedPnl += _unrealizedPnl;
}
```

Therefore, if we assume that a user with more positions and exposure to more tokens needs to tweak their positions from time to time, and the number of actions correlates the number of positions, the gas usage really scales quadratically to the number of positions for such a user.

A2	Suggestion regarding the numerical methods of CurveMath.vy	**ACKNOWLEDGED**

[Below we use the notation of the curve crypto whitepaper]

The CurveCrypto invariant in the case of pools with only two assets (N=2) can be simplified into a low degree polynomial, which could lead to a faster convergence of the numerical methods.

The coefficient K, when N=2 (we denote by x and y the deposits of the two assets in the pool), is given by the formula

$$K = \frac{4A\gamma^2 xy(x+y)D^2}{(D^2(\gamma+1)-4xy)^2}$$

If we multiply both sides of the equation $F(x, D) = 0$ by the denominator of K we get an equivalent equation, which is polynomial in all three variables x, y and D:

$$4A\gamma^2 xy(x+y-D)+(D^4(\gamma+1)^2+16x^2y^2-8xyD^2(\gamma+1))(xy-\frac{D^2}{4})=0$$

As you can see it is a cubic equation for x and y and you can use the formulas for cubic equations either to compute faster the solution or to get a better initial value for the iterative method you are currently using. We believe it would be worth spending some time experimenting with the numerical methods to get the fastest possible convergence (and consequently reduced gas fees paid by the users).

A3	No functionality to remove AMMs	**ACKNOWLEDGED**

Governance has the ability to whitelist AMMs via `ClearingHouse::whitelistAmm` method, while there is no functionality to remove or blacklist an AMM.

A4	`MarginAccount` collateral index checks are missing	**ACKNOWLEDGED**

There are several external methods of `MarginAccount`, namely `addMargin`, `addMarginFor`, `removeMargin`, `liquidateExactRepay` and `liquidateExactSeize` that do not implement a check on the collateral index supplied, which can lead to the ungraceful termination of the transaction if an incorrect index has been supplied. A simple check such as:

```
require(idx < supportedCollateral.length, "Collateral not supported");
```

could be used to also inform the user of the problem with their transaction.

A5	AMM::PositionChanged event is missing timestamp field	**ACKNOWLEDGED**

The `AMM::PositionChanged` event is potentially missing a timestamp field that all other trader related events (`LiquidityAdded`, `LiquidityRemoved`, `Unbonded`) incorporate.

A6	Commented out code	**ACKNOWLEDGED**

In method `MarginAccount::isLiquidatable` the following line is commented out:

```
_isLiquidatable = IMarginAccount.LiquidationStatus.IS_LIQUIDATABLE;
```

This is because `IMarginAccount.LiquidationStatus.IS_LIQUIDATABLE` is equal to 0, which will be the default value of `_isLiquidatable` if no value is assigned to it, thus the above assignment is not necessary. Nevertheless, explicitly assigning the enum value makes the code much more readable and intiutive.

A7	Magic constants	**ACKNOWLEDGED**

There are several magic constants throughout the codebase, many of them related to the precision of token amounts, making it difficult to reason about the correctness of certain computations. The developers of the protocol are aware of the issue and claim that they have developed extensive tests to make sure nothing is wrong in this regard.

A8	Oracle price decimals assumption	**ACKNOWLEDGED**

The Oracle contract code makes the assumption that the price value returned by the ChainLink oracle has 8 decimals. This assumption appears to be correct if the oracles used report the price in terms of USD. Nevertheless, using the oracle's available `decimals` method and avoiding such a generic assumption would make the code much more robust.

A9	Code can be reused	**ACKNOWLEDGED**

The following code shared by methods `MarginAccount::liquidateExactRepay` and `MarginAccount::liquidateExactSeize` can be factored out in a separate method and reused:

```
clearingHouse.updatePositions(trader); // credits/debits funding
LiquidationBuffer memory buffer = _getLiquidationInfo(trader, idx);
if (buffer.status != IMarginAccount.LiquidationStatus.IS_LIQUIDATABLE) {
    revert NOT_LIQUIDATABLE(buffer.status);
}
```

In addition, all the code of `AMM::isOverSpreadLimit`:

```
function isOverSpreadLimit() external view returns(bool) {
    if (ammState != AMMState.Active) return false;

    uint oraclePrice = uint(oracle.getUnderlyingPrice(underlyingAsset));
    uint markPrice = lastPrice();
    uint oracleSpreadRatioAbs;
    if (markPrice > oraclePrice) {
        oracleSpreadRatioAbs = markPrice - oraclePrice;
    } else {
        oracleSpreadRatioAbs = oraclePrice - markPrice;
    }
    oracleSpreadRatioAbs = oracleSpreadRatioAbs * 100 / oraclePrice;

    if (oracleSpreadRatioAbs >= maxOracleSpreadRatio) {
        return true;
    }
    return false;
}
```

except line `uint markPrice = lastPrice();` can be factored out in another method, e.g., `_isOverSpreadLimit(uint markPrice)`, which will have `markPrice` as an argument. Then method `_isOverSpreadLimit` can be reused in methods `_short` and `_long`.

A10	Access modifiers	ACKNOWLEDGED
Methods `syncDeps` of MarginAccount and InsuranceFund could be declared `external` instead of `public`.		
A11	Unused code/contracts	ACKNOWLEDGED
tests/Executor.sol is not used.		
A12	Compiler known issues	INFO
The contracts were compiled with the Solidity compiler v0.8.9 which, at the time of writing, have some known bugs. We inspected the bugs listed for this version and concluded that the subject code is unaffected.		

CENTRALIZATION ASPECTS

As is common in many new protocols, the owner of the smart contracts yields considerable power over the protocol, including changing the contracts holding the user's funds, adding AMMs and tokens, which potentially means borrowing tokens using fake collateral, etc.

In addition, the owner of the protocol can:

- Blacklist any user.
- Set important parameters in the vAMM which change the price of any assets: `price_scale`, `price_oracle`, `last_prices`. This allows the owner to potentially liquidate otherwise healthy positions or enter into bad debt positions.

The computation of the Margin Fraction takes into account the weighted collateral, whose weights are going to be decided by governance. Currently the protocol uses NFTs for governance but in the future the decisions will be made through a DAO. Currently, there is no relevant implementation, i.e., the Hubble protocol does not yet offer a governance token. Still, even if the final solution is decentralized, governance should be really careful and methodical when deciding the values of the weights. We believe that another, safer approach would be to alter these weights in a specific way defined by predetermined formulas and allow only small adjustments by the DAO.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure the most prominent protocols in the space. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Prominent blockchain protocols hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.