Marginal Protocol Security Review

Auditors

0xLeastwood, Lead Security Researcher

Desmond Ho, Lead Security Researcher

Jeiwan, Security Researcher

Jonatas Martins, Associate Security Researcher

Report prepared by: Lucas Goiriz and Jonatas Martins

January 9, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Marginal is a decentralized, permissionless, peer-to-pool AMM for physically settled perpetual on any pair of fungible tokens

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Marginal Protocol according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Marginal Protocol engaged with Spearbit to review the v1-core protocol. In this period of time a total of **32** issues were found.

Summary

Project Name	Marginal Protocol
Repository	v1-core
Commit	47e413...11a955
Type of Project	AMM, Perpetuals
Audit Timeline	Dec 4 to Dec 15
Two week fix period	Dec 18 - Dec 29

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	4	2	2
Low Risk	10	5	5
Gas Optimizations	9	6	3
Informational	8	5	3
Total	**32**	**19**	**13**

5 Findings

5.1 High Risk

5.1.1 Exact output swaps pay less fees than exact input swaps

Severity: High Risk

Context: MarginalV1Pool.sol#L728, MarginalV1Pool.sol#L768

Description: The `MarginalV1Pool.swap()` function allows performing exact input and exact output swaps. In exact input swaps users specify the exact amount of tokens to sell; the output amount is computed by the contract. In exact output swaps users specify the exact amount of tokens to buy; the input amount is computed by the contract.

The `swap()` function takes swap fees from the input amount:

1. For exact input swaps, the fee percentage is applied to the user-specified input amount (MarginalV1Pool.sol#L691-L695):

```
bool exactInput = amountSpecified > 0;
int256 amountSpecifiedLessFee = exactInput
    ? amountSpecified -
        int256(SwapMath.swapFees(uint256(amountSpecified), fee))
    : amountSpecified;
```

2. For exact output swaps, the fee percentage is also applied to the input amount (MarginalV1Pool.sol#L766-L769):

```
uint256 fees0 = exactInput
    ? uint256(amountSpecified) - uint256(amount0)
    : SwapMath.swapFees(uint256(amount0), fee);
amount0 += int256(fees0);
```

However, the input amount in exact output swaps is computed as an *after fee* amount (MarginalV1Pool.sol#L709-L714):

```
// amounts without fees
(amount0, amount1) = SwapMath.swapAmounts(
    _state.liquidity,
    _state.sqrtPriceX96,
    sqrtPriceX96Next
);
```

Thus, the fee amount will be smaller than expected because an after fee amount is always smaller than the respective before fee amount.

As a result, exact output swaps are cheaper for traders, which causes a loss of earnings for liquidity providers and gives financial benefit to abusers of the miscalculation.

Recommendation: When computing swap fees in exact output swaps, consider using this formula: `(input amount * fee) / (1e6 - fee)`. This change should be applied to both "zero for one" and "one for zero" swaps.

Marginal: Agreed, fixed in commit 8e21eff8.

Spearbit: Fixed.

5.2 Medium Risk

5.2.1 Liquidation rewards are not robust enough to incentivize edge case liquidations

Severity: Medium Risk

Context: MarginalV1Pool.sol#L307-L380, Position.sol#L233-L240

Description: Liquidation rewards are necessary to help incentivize liquidations on unhealthy positions. Tokens are set aside upon opening a position and released when the trader settles their debt. The current implementation allocates a fixed percentage of the position size to the liquidation reward. However, the protocol does not enforce a minimum position and therefore it is not possible to determine if the value of this liquidation reward is sufficient to cover the gas costs of calling the `liquidate()` function.

There are a few other limitations to the current approach:

- A fixed percentage of token amount does not account for the value of each token and it's underlying precision.
- Minimum position sizes are not enforced, leading to potential dust positions.
- The value of the liquidation reward scales linearly when it should be a fixed cost paid by the trader.
- When a trader's position is eligible for liquidation, the value of their liquidation reward has also decreased by the same proportion, this is not ideal.

Ultimately, liquidation rewards can be allocated in native ether and calculated by taking the upper bound for what the gas price is expected to reach under pool conditions and the worst-case scenario gas cost for calling the `liquidate()` function.

Recommendation: It is important that liquidation rewards are paid in the native token of the chain for which the marginal pool is deployed. An upper bound for future gas prices can be used to provide significant coverage for where liquidations should be seamless.

Although, it seems difficult to maintain protocol liveness when gas prices are extraordinarily high during a time where there is significant volatility in the 24 hour TWAP oracle. Ultimately, in these cases, LPs would be incentivized to liquidate positions and free up their own liquidity. But there may need to be some more thought around this approach.

Marginal: Fixed in commit c7feb992.

Used `block.basefee` with min base fee constant of 40 Gwei for ETH mainnet. 150K gas cost constant for liquidate, as estimates in main net-fork on Uniswap v3 ETH/USDC 30 bps pool in integration tests show `liquidate` can get up to 120K (searching through oracle observations array is variable).

Incentivized liquidators via a premium of 2x current base fee (so min 80 Gwei), which ensures likely liquidation for most blocks even if not during initial gas spike.

Spearbit: Verified fix. Something to consider when calculating the minimum liquidation reward off `block.basefee` and expecting sufficient `msg.value` to be sent in. Upon signing a transaction and waiting for it to be included in a future block. The block base fee may decrease and the trader effectively ends up transferring in too much. Is it fair to not refund any excess ether here?

Marginal: Agreed on the excess worries, but was going to have this responsibility delegated to `v1-periphery` contract in NFT position manager. Basically calculate the reward using the position lib on periphery, and if `msg.value` has excess, call `refundETH()` on manager via multi call or simply sweep it back to original sender.

Spearbit: Acknowledged, this will be handled by the `v1-periphery` contracts.

5.2.2 Pool positions may have excess margin collateral removed when unsafe

Severity: Medium Risk

Context: MarginalV1Pool.sol#L236-L471, Position.sol#L367-L397

Description: Pool positions have their margin requirements calculated on price data before the position was opened. If excess collateral was provided as part of the position's margin amount, then this collateral will contribute to the healthiness of a trader's position by collaterizing the debt owed in the form of the other token.

Pool position safety is checked by taking the 12 hour TWAP `oracleSqrtPriceX96` price to ensure `liquidityCollateral >= liquidityDebt`. If this constraint is not upheld, then the position is eligible for liquidation and LPs will receive back at least the liquidity locked in the trader's position. Additional liquidity is earned by LPs in the from of any leftover margin.

Minimum margin requirements are *always* calculated using `sqrtPriceX96 = TickMath.getSqrtRatioAtTick(position.tick)` which is set when the position date is assembled.

```
function marginMinimum(
    Info memory position,
    uint24 maintenance
) internal pure returns (uint128) {
    uint160 sqrtPriceX96 = TickMath.getSqrtRatioAtTick(position.tick); // price before open
    if (!position.zeroForOne) {
        // cx >= (1+M) * dy / P - sx
        uint256 debt1Adjusted = (uint256(position.debt1) *
            (1e6 + maintenance)) / 1e6;

        uint256 prod = sqrtPriceX96 <= type(uint128).max
            ? Math.mulDiv(
                debt1Adjusted,
                FixedPoint192.Q192,
                uint256(sqrtPriceX96) * uint256(sqrtPriceX96)
            )
            : Math.mulDiv(
                debt1Adjusted,
                FixedPoint128.Q128,
                Math.mulDiv(sqrtPriceX96, sqrtPriceX96, FixedPoint64.Q64)
            );
        return
            prod > uint256(position.size)
                ? (prod - uint256(position.size)).toUint128()
                : 0; // check necessary due to funding
    } else {
        // cy >= (1+M) * dx * P - sy
        uint256 debt0Adjusted = (uint256(position.debt0) *
            (1e6 + maintenance)) / 1e6;

        uint256 prod = sqrtPriceX96 <= type(uint128).max
            ? Math.mulDiv(
                debt0Adjusted,
                uint256(sqrtPriceX96) * uint256(sqrtPriceX96),
                FixedPoint192.Q192
            )
            : Math.mulDiv(
                debt0Adjusted,
                Math.mulDiv(sqrtPriceX96, sqrtPriceX96, FixedPoint64.Q64),
                FixedPoint128.Q128
            );
        return
            prod > uint256(position.size)
                ? (prod - uint256(position.size)).toUint128()
                : 0; // check necessary due to funding
```

```
    }
}
```

However, an unsafe position under certain circumstances may still allow a trader to `adjust()` their margin and remove some `marginDelta` amount. This allows them to re-coup some of their collateral before LPs are able to claim back locked liquidity which is owed to them. Fortunately, the insurance invariants enforced by the protocol ensure sufficient collateral is set aside that guarantees LPs receive at least their locked liquidity back once a position expires and is either settled or liquidated.

Recommendation: Consider adding `position.safe()` checks to the end of `open()` and `adjust()` calls to ensure the protocol is better protected against any potential bad debt accrual. Adding this check to the `open()` function is important because of potential price lag between the spot price and the 12 hour TWAP price.

If the `sqrtPriceX96` spot price has recently improved, then long/`zeroForOne` positions may be eligible for liquidation as soon as their position has been opened because the univ3 TWAP oracle is slow to react to short term price movements.

Marginal: Agreed for better UX from trader perspective, there should be a `position.safe()` check in `open()` and `adjust()` to ensure they don't get immediately liquidated post-call.

However, bad debt shouldn't accrue even in this scenario due to the insurance mechanism + margin minimum enforcement. More that the position owner gets unexpectedly liquidated post-call as margin must satisfy (in one for zero case):

```
c_x (safe) >= (1 + M) * d_y / min(P, TWAP) - s_x
```

to be safe from liquidation.

To prevent extra gas costs at core level + contract size increases, the plan is to resolve these trader UX issues in `v1-periphery` by having UI query `v1-periphery/lens/Quoter.sol` -- which should return whether position *would* be safe post open, adjust.

Spearbit: Acknowledged. Additional validation will be added to the `v1-periphery` contracts. The insurance mechanism already prevents any bad debt accrual.

Marginal: Quick fix in commit bdeb55c and ed93918.

Spearbit: Acknowledged the addressed fixes in `v1-periphery` contracts, which aren't covered in this review

5.2.3 `sqrtPriceX96` can be initialized to put a pool in an invalid state

Severity: Medium Risk

Context: MarginalV1Pool.sol#L184-L199, MarginalV1Factory.sol#L59-L97

Description: Marginal pools are deeply inspired by univ2 and univ3 mechanics, namely, pools are priced according to the same tick math as in univ3. However, LPs mint/burn tokens across the entire range of possible prices (from the lowest tick to the upper-most tick), which is similar to how liquidity is allocated in univ2.

Because marginal pools do not allow for LPs to provide liquidity for specific tick ranges, the initial `sqrtPriceX96` is extremely important when it comes to the pricing of LP tokens.

The current process for pool deployments is generally permissionless, except the owner of the factory contract has control over the available leverage ratios which pools can be deployed under. Therefore, there exists only one pool for each `token0`, `token1`, `uniswapV3Fee` and `maintenance` parameters.

Malicious actors can ensure unique pools are initialized to an extremely high/low `sqrtPriceX96`, making it incredibly risky for anyone to provide liquidity as their funds would instantly be arbitraged across the target univ3 pool for which marginal uses as an oracle for it's 12 hour TWAP.

Note:

> While the implementation is generally equivalent to univ3 pool deployments, LPs can choose provide liquidity across specific tick ranges, allowing them to be exposed only to certain price movements. The

initial value for `sqrtPriceX96` is purely informative and helps facilitate future swaps once liquidity has been added to the pool. When a swap does happen, it will continue to iterate through ticks until there is available liquidity, allowing for efficient price discovery. While univ3 supports single-sided liquidity, marginal protocol should not, but by impacting the initialized price, it can be possible to provide single-sided liquidity due to severe precision truncation by the other pool token.

Recommendation: It is important that the initial pricing is estimated closely to the spot price on univ3. Ideally, someone needs to bootstrap liquidity on the pool and take on some risk when initializing the `sqrtPriceX96` price. Assuming it is within some bounds of the oracle's 12 hour TWAP, then arbitrageurs would be able to bring the pricing back to some fair value.

This fix could be made within the `initialize()` function whereby a minimum amount of liquidity must first be provided. It would however be difficult to standardize this minimum amount as tokens vary in precision and in value. So ultimately, arbitrageurs need some incentive to perform the arbitrage if there is any price drift.

Marginal: Agreed. Fixed in commit 2ba7e61a.

Pools now initialized on first call to `mint()`, with `initialize()` as a private function. On initialize, state is initialized to the `oracleSqrtPriceX96` value fetched from the oracle averaged over `secondsAgo`. Pools now have `unlocked` set to `2` or true on deploy, with the understanding that calling all other functions before calling `mint()` should revert.

On initial `mint()`, `MINIMUM_LIQUIDITY` shares of liquidity are locked in the pool contract to ensure always a minimum of amount of liquidity is available for swaps, in case price diverges significantly due to all LPs removing liquidity. This available liquidity for swaps should prevent stuck states for the next LP to easily mint at (or close to) the spot price.

Further, if the oracle price is significantly different from the current spot price at initialization, LPs can simply provide the minimum liquidity then swap to adjust price to spot, before providing the full amount of liquidity they intend to provide. We can have a helper for this on the periphery.

Spearbit: I think this is a much more reasonable design choice. Similar to univ2, liquidity is being minted and effectively burnt upon initialization, ensuring the price resemble's that of it's univ3 pool oracle.

My question is this `MINIMUM_LIQUIDITY` sufficient enough to prevent significant price manipulation? i.e. it may still make sense for someone to mint the initial liquidity and wait for a new LP position to be created. This user can be front-ran and forced to mint LP tokens at an unfair price and then the attacker can back-run the trade for profit. Although, I guess the mint callback should have some validation on the user side but in most cases I would not expect this.

Marginal: Yea agreed. `MINIMUM_LIQUIDITY` is super small, so price can move dramatically via a simple swap at these levels, but viewing it as almost a good thing in the sense that for dust amounts, pool state can't get into a stuck state anymore.

I think we can check on the mint callback during pool creation for any unfair price manipulation, probably user specified bounds or even swapping to the oracle pool price itself before the pool creator provides significant funds to LP with -- basically in my mind to be super safe, could have the pool initialization helper adapted to have pool creator mint `MINIMUM_LIQUIDITY` at the 12hr TWAP price which lags the `oracle` spot price, swap to the actual oracle price with user specified bounds, then LP full amount they wanted to mint to Marginal pool.

I'm not sure there's any way around potentially moving pool to any price they want, to be honest. But at least in current implementation, the pool can't get stuck. So other users can easily swap Marginal price back to actual spot oracle price using dust amounts, with minimal cost if they have to.

Spearbit: Verified fix. Price manipulation is expected behaviour for newly deployed pools with minimal liquidity. The pool state will still maintain it's working state which is what's being prioritised in this fix.

5.2.4 Frequent position adjustments could cause zero funding debt accumulation

Severity: Medium Risk

Context: Position.sol#L323-L333, Position.sol#L338-L341

Description: Between time periods, the debt accumulated is calculated as

$$d_{y_{t+\tau}} = d_y \cdot (1.0001^{(a_{t+\tau}-a_t)/\tau-(b_{t+\tau}-b_t)/\tau})^{\tau/T}$$

$$= d_y \cdot (1.0001^{(a_{t+\tau}-a_t)-(b_{t+\tau}-b_t)})^{1/T}$$

$$= d_y \cdot (1.0001^{\frac{2[(a_{t+\tau}-a_t)-(b_{t+\tau}-b_t)]}{T}})^{\frac{1}{2}}$$

for time τ since the last funding sync. We see that if `arithmeticMeanTick` = $\frac{2[(a_{t+\tau}-a_t)-(b_{t+\tau}-b_t)]}{T}$ is zero, $d_{y_{t+\tau}} = d_y$, ie. there will be no change in debt accumulation. This would be achieved if `delta < fundingPeriod / 2`.

Hence, the maximum τ allowable for zero debt accumulation would be `fundingPeriod / (2 * tickDiff)` where `tickDiff` is the average tick difference between the marginal and UniV3 spot TWAPs within τ.

Note that the maximum `tickDiff` is capped by `tickCumulativeRateMax`. With its value set at `920` and `fundingPeriod = 604800`, in the best case where the average tick difference exceeds `tickCumulativeRateMax`, a position can wait at most `604800 / (2 * 920) = 328s` (~5 mins) to accumulate zero debt. Smaller relative tick differences would allow larger τ values for zero debt accumulation.

Recommendation: The objective would be to minimise `fundingPeriod / (2 * tickDiff)`. Hence, it would make sense to either reduce `fundingPeriod` and / or increase `tickCumulativeRateMax`. The latter would mean raising the funding cap.

Marginal: Agreed, but the tradeoff to a smaller `fundingPeriod` would be increased risk from oracle manipulation of the position safety condition (see scratch math below). Would be comfortable reducing the `fundingPeriod` to minimum 48 hours (2 days), which gives the attacker an extra 25% "boost" to the effect of manipulating the oracle on the safety condition vs figures calculated in the oracle manipulation appendix. Although for ETH L1 mainnet, would prefer to leave as is.

- **Mathematical reasoning**

The position safety condition is

$$(c_x + s_x)P_{O_{t-a,t}} \geq (1 + M)d_{y_t}$$

where a is `secondsAgo` to avoid confusion with tau above. $P_{O_{t-a,t}}$ is the oracle TWAP averaged over `secondsAgo`.

As mentioned, the debt update for funding is

$$d_{y_t} = d_{y_{t-\tau}} \left(\frac{P_{M_{t-\tau,t}}}{P_{O_{t-\tau,t}}} \right)^{\tau/T}$$

where $P_{M_{t-\tau,t}}$ is the Marginal pool TWAP averaged over τ since the last sync. Plugging into the safety condition, you can start to see that the funding sync poses increased risk of attack due to the additional occurrence of the oracle TWAP the manipulator is moving -- helps the attacker out, so less capital required to move actually.

If my math is correct, I find that roughly to move the safety condition δ ticks (i.e. deviation of 1.0001^{δ}) requires a spot tick change of

$$\delta = \frac{\tau}{a}\left(1 + \frac{a}{T}\right)\left(i_t - i_0\right)$$

where i_t is the spot tick attacker moves the Uniswap spot price to and i_0 is the unmanipulated tick prior.

Currently we have $a = 1/2$ day and $T = 7$ days, which means funding doesn't reinforce the oracle manipulation too much (about 7% easier tick wise). As $T \to a$, we start reducing the upfront capital required to attack the oracle.

If we set $T = a = 1/2$ day for a Uniswap spot pool with $1M in full range liquidity and manipulate in the two block attack for *tau* = 12 seconds, the $4B upfront capital requirement to move the oracle 50 bps reduces instead to $45M upfront to move the oracle 25bps but reinforced by a funding sync that pushes the safety condition another 25 bps in the attacker's advantage to 50 bps total (roughly if scratch math isn't wrong).

Spearbit: Acknowledged that there is a significant drawback in reducing `fundingPeriod` of lower capital requirement in oracle manipulation. Choosing the lesser of 2 evils, reducing to minimally 2 days or perhaps half a week would alleviate the problem whilst having a realistically sufficient high capital requirement for oracle manipulation.

5.3 Low Risk

5.3.1 Oracle observation cardinality may be insufficient and revert during liquidation

Severity: Low Risk

Context: MarginalV1Pool.sol#L213-L221

Description: Uniswap V3's oracle provides price and liquidity data for external use. This data is written as an observation into a circular array at the start of each block in which the pool was interacted with. Within marginal, this data is used to find the 12 hour TWAP used to price positions upon liquidation. In other parts, the protocol will always, in some way, need to retrieve the oracle tick cumulative in order to sync the trader's position. Therefore, the liveness of each pool is tightly coupled with the functionality of univ3's oracle implementation.

The current check in the constructor does indeed check if the current array of historical observations is able to support it's TWAP pricing on pool deployment. But this in no way guarantees the same behavior afterwards. If we consider an active univ3 pool to have a pool interaction in each block, then an observation will be made once at the beginning of the block and continue to fill the oracle array until the cardinality is reached and then wrap around and start writing over old data.

It's noted that the pool factory contract performs the following check upon deployment:

```
(, , , uint16 observationCardinality, , , ) = IUniswapV3Pool(oracle)
    .slot0();
if (observationCardinality < observationCardinalityMinimum)
    revert InvalidObservationCardinality(observationCardinality);
```

The check asserts that the pool has some minimum oracle array size. However, the true array size is `observationCardinalityNext` which includes already allocated space.

Even though LPs are incentivised to grow the oracle array, this costs some protocol liveness because the oracle takes several hours to grow into it's new length. Effectively, if the pool cannot support the 12 hr TWAP under its worst case scenario, then liquidations can be prevented for up to 12 hours as the univ3 pool needs time to fully support this.

Recommendation: Consider updating the pool cardinality check in the factory contract to compare `observationCardinalityNext` instead of `observationCardinality` and provide some guarantees in support of oracle liveness by ensuring univ3 pools support the 12 hr TWAP under extreme cases.

Marginal: The pool deployment check is already conservative and `observationCardinalityNext` may cause the price query to fail if the index hasn't yet been pushed far enough to effectively bump the cardinality to `observationCardinalityNext`.

Spearbit: Acknowledged as a won't-fix. I agree that checking against `observationCardinalityNext` instead of `observationCardinality` may cause some temporary liveness issues. It makes sense to be conservative upon deployment. Although, it is important to ensure that `observationCardinalityMinimum` is set to some value that represents oracle writes on each block for the entire duration of the TWAP.

5.3.2 Funding payments may decrease overall system debt

Severity: Low Risk

Context: Position.sol#L315-L365

Description: Marginal protocol has a funding rate mechanism inspired by Opyn's Squeeth product. Upon opening a position with the protocol, the `position.assemble()` call takes a snapshot of `tickCumulativeDelta` which represents the delta between univ3's TWAP tick cumulative and marginal's tick cumulative.

A funding rate is charged whenever a trader's position is synced, updating the trader's nominal debt owed to the protocol. This rate is clamped by funding rate bounds (approximately 10% per week) and is paid by one side and received by the other. Ultimately, if the pool is long heavy, then traders who are long have their debt increased and in a similar fashion, short traders have their debt reduced.

However, it is important to note that positions are synced at any point in time the trader interacts with the pool. These include the `adjust()`, `settle()` and `liquidation()` actions. Therefore, traders who are set to have their debt reduced are incentivized to sync their positions and the other side is not.

Additionally, funding payments do not consider the difference in pool weightings between long and short traders. An assumption is made in that any price deviance is caused only by the protocol's traders, which does not consider swaps made on the univ3 and marginal pools.

These swaps propagate to the lagging oracle price and used to determine the current tick cumulative delta. As a result, there are possible pool states where system debt is being reduced, albeit at a slow rate. Fortunately, the insurance allocated on new positions avoids any impact on LPs.

Recommendation: Consider re-designing how funding payments are calculated by taking the overall debt on each side of the pool. The debt change between the two sides should be zero-sum and not prone to manipulation. This also avoids issues where some traders are not incentivized to sync their debt so the funding rate is paid favorable. However, there is likely some game theory that would prevent this behavior.

Alternatively, it may be simpler in design to have one-way funding payments. In that, system debt is only ever increased to incentivize certain traders to settle their position or be liquidated.

Marginal: Acknowledged, but going to not change for v1 given want to be consistent with funding in kind for the interest rate so positions are perp-like.

One way funding is definitely interesting, but would eliminate the possibility of basis trade strategies from trader perspective, which seems like a disadvantage. Would be interesting to redesign funding payments so zero-sum, like a traditional CEX perp, but not sure how we best do this in a passive liquidity pool AMM model when traders expect to be able to take out positions at any moment (i.e. would expect skew in the passive AMM model given don't have a 1:1 in trader counterparties).

Spearbit: Acknowledged.

5.3.3 Marginal pools may have self-reflexive oracle dependencies

Severity: Low Risk

Context: MarginalV1Pool.sol

Description: Marginal pool token pricing closely follows the spot pricing for the same tokens in univ3 as any price deviance quickly incentivizes arbitragers to take action.

Considering marginal pools are typically geared towards speculators, it is likely that univ3 pools remain the canonical pool when it comes to token pricing. In the event where available liquidity in the marginal pool is greater than that of all available liquidity found externally, it no longer becomes possible for arbitrageurs to always close-out any price deviance.

While prices may converge to a single point, the convergence is concentrated around liquidity and therefore if marginal attracts the majority of liquidity, then it also becomes the canonical price. Because we are dealing with an external oracle reliant on the univ3 pool, it is possible that marginal pools become self-reflexive and any price

action in the marginal pool is later reflected by the univ3 pool and this later propagates back via the 12 hour TWAP oracle.

It is unclear on what the impact of this behaviour would be, but it potentially leads to more predictable oracle behaviour that is prone to manipulation.

Recommendation: Consider simulating marginal/univ3 pools of varying liquidity and how prices may converge under different liquidity conditions. It is important that the pool state is monitored to check if this ever happens in real-time.

Marginal: Agreed. We plan to have simulations before launch. We'll try to look at this case as well.

Spearbit: Acknowledged. This will be actively explored prior to launch.

5.3.4 `adjust()` must always check minimum margin requirements due to funding rate payments

Severity: Low Risk

Context: MarginalV1.sol#L123

Description: Minimum margin amounts scale with the funding rate which is further determined by pool utilization. In order to incentivize price convergence to a TWAP price that is in-line with the univ3 oracle price, a trader's position debt is increased if they are long X token and marginal's TWAP price is greater than the univ3 oracle TWAP price. This pushes the trader's position closer to liquidation, incentivising early settlement and the return of pool liquidity which in turn facilitates better pricing in marginal pools.

When a trader's position is synced and it's debt is updated, the minimum margin requirement will also be increased. As a trader goes to `adjust()` their position and add collateral, the function will skip any checks that the minimum margin requirement has been met. It is therefore important to check that the new margin amount is sufficient regardless if margin is being removed or added.

Recommendation: Perform minimum margin checks regardless if margin is being removed or added. This should account for any increases in debt due to the funding rate, forcing the trader to add sufficient margin or risk being liquidated.

Marginal: Agreed. Fixed in commit bdba9a9d.

Spearbit: Fixed.

5.3.5 Newly opened positions should check both token debts are non-zero

Severity: Low Risk

Context: MarginalV1Pool.sol#L292-L295

Description: Liquidity pools are made up of two tokens which may vary in precision but `sqrtPriceX96` generally computes in terms of the token with the least precision. Therefore, if we open a long position with dust amounts of Y provided, then it's entirely possible this `debt0` amount calculated during position creation is truncated along with `insurance0`.

Recommendation: To better protect against this edge case, consider updating the referenced code block to the following:

```
if (
    position.size == 0 ||
    position.debt0 == 0 || position.debt1 == 0
) revert InvalidPosition();
```

This should both debts are non-zero and protect against any potential rounding due to varying precision in the two pool tokens.

Marginal: Fixed in commit 6c623173.

Spearbit: Fixed.

Marginal: Further fixed in commit 6d0690f.

Imposes `MINIMUM_SIZE` constraint on `position.size`, `debt0`, `debt1`, `insurance0`, `insurance1` given truncation issues in invariant fuzz tests for dust positions.

`MINIMUM_SIZE = 10000` on position amounts should be fine for most token pairs. Only place where could cause issues I'm aware of would be `WBTC` at 8 decimals for requiring larger position sizes. But at $1M per WBTC, min size would be $100 in notional, so seems fine.

Spearbit: Verified fix. Minimum position size is now enforced across all position data types.

5.3.6 Use `Ownable2Step` for access control

Severity: Low Risk

Context: MarginalV1Factory.sol

Description: The `MarginalV1Factory` contract executes a single-step ownership change, which may be prone to accidental transfers of access control. The Ownable2Step variant of the Ownable contract can protect such accidental transfers.

Recommendation: It is recommended to use `Ownable2Step` for access control in `MarginalV1Factory` contract.

Marginal: Agree. But `owner` will be set to a timelock controller, so not going to implement this in core contracts.

Spearbit: Acknowledged.

5.3.7 Collecting fee may revert when transfer zero amount

Severity: Low Risk

Context: MarginalV1Pool.sol#L918-L925

Description: The `collectProtocol()` function checks if `protocolFees.token0 == 0 || protocolFees.token1 == 0`. After the first iteration, `amount0` and `amount1` are calculated from `protocolFees.token0 - 1` and `protocolFees.token1 - 1` respectively, always leaving one token of each. In a subsequent call where `amount0 == 1` and/or `amount1 == 1`, the amount transferred to the recipient would be zero. The issue arises because some tokens, such as `LEND`, may revert to zero-value transfer. As a result, it would be necessary to accrue protocol fees before calling the `collectProtocol` function again.

Recommendation: It is recommended to check if `amount0` or `amount1` is different from zero before transfer it

```
+ if(amount0 != 0){
    protocolFees.token0 -= amount0;
    TransferHelper.safeTransfer(token0, recipient, amount0);
+ }

+ if(amount1 != 0){
    protocolFees.token1 -= amount1;
    TransferHelper.safeTransfer(token1, recipient, amount1);
+ }
```

Marginal: Agree `amount0 = amount1 = 0` poses issues, but given `onlyFactoryOwner` can call this function to collect fees, governance can wait until `amount0 > 0 && amount1 > 0` to call. Going to leave it as is for now, given the contract bytecode size increases significantly with the suggested change (at least for now).

Deployment Bytecode sizes before:

Contract	Bytecode size
MarginalV1PoolDeployer	- 24,547B (99.88%)
MarginalV1Pool	- 24,093B (98.03%)

Deployment Bytecode sizes after:

Contract	Bytecode size
MarginalV1PoolDeployer	- 24,569B (99.97%)
MarginalV1Pool	- 24,115B (98.12%)

Spearbit: Given the significant increase in contract bytecode size with the changes, and since the function call is permissioned, we have agreed to leave it as is for now. Acknowledged.

5.3.8 Token amounts to be provided by LPs should be rounded up

Severity: Low Risk

Context: LiquidityMath.sol#L13-L22

Description: `LiquidityMath.toAmounts()` is used to calculate the token amounts required to be sent to / given by the liquidity provider when removing / adding liquidity. When minting shares (adding liquidity), the amount pulled should be rounded up, as suggested by the ERC-4626 spec. We also see this as in UniV3's `getAmount0Delta()` and `getAmount1Delta()` functions that rounds up / down when liquidity is added / removed from the pool.

Recommendation: Round up the token amounts to be provided by LPs for adding liquidity.

Marginal: Agreed.

Fixed in commit c37b7fec. Approach to round up is rough to avoid contract bytecode size exceeding limits, so slightly unfair to LPs (but not that bad really given 1 Wei).

Contract size decreases from `100.04%` on deployer to:

Contract	Deployment Bytecode Sizes
MarginalV1PoolDeployer	- 24,559B (99.93%)
MarginalV1Pool	- 24,105B (98.08%)

Spearbit: Fixed. Given the limitation on the contract bytecode size, a standardized addition of 1 wei to both `token0` and `token1` amounts was added.

5.3.9 After swap price calculation can overflow

Severity: Low Risk

Context: SqrtPriceMath.sol#L66, SqrtPriceMath.sol#L104

Description: The `SqrtPriceMath.sqrtPriceX96NextSwap()` function is used to compute the price after swapping an amount of tokens (`amountSpecified`). The function handles four cases and, in two of them, multiplies `amountSpecified` by `2**96` (the bitwise shifting by `FixedPoint96.RESOLUTION` is identical to multiplying by `2**96`):

1. SqrtPriceMath.sol#L64-L67:

```
nextX96 =
    sqrtPriceX96 +
    (uint256(amountSpecified) << FixedPoint96.RESOLUTION) /
    liquidity;
```

2. SqrtPriceMath.sol#L102-L105:

```
nextX96 =
    sqrtPriceX96 -
    (uint256(-amountSpecified) << FixedPoint96.RESOLUTION) /
    liquidity;
```

Keeping in mind that the maximal integer value in EVM is `2**256-1`, the above multiplications can overflow when the value of `amountSpecified` is greater than `2**160-1`: multiplying `2**160` by `2**96` results in a value that's greater than `2**256-1`.

As a result, an attempt to swap an amount that's greater than `2**160-1` will revert.

Recommendation: In cases when `amountSpecified` equals or is smaller than `type(uint160).max` (`2**160-1`) consider keeping using the current implementation. In cases when `amountSpecified` is greater than `type(uint160).max`, consider using `Math.mulDiv()`. Handling the two cases differently will save gas for the majority of users since `Math.mulDiv()` consumes more gas but the majority of token amounts are less than `type(uint160).max`.

Marginal: Fixed in commit 11e82b5e. The fix is re-applied in commit 6689416.

Spearbit: Fixed.

5.3.10 Missing indexed fields in an event

Severity: Low Risk

Context: MarginalV1Factory.sol#L22-L26

Description: In `MarginalV1Factory`, the `PoolCreated` event misses indexed fields. The even is emitted whenever a new Marginal pool is created (MarginalV1Factory.sol#L96), however the absence of indexed fields will make it harder for monitoring and analysis tools to discover created pool.

Recommendation: In the `PoolCreated` event, consider indexing `token0`, `token1`, and `oracle` fields.

Marginal: Agree, fixed in commit a1483cda.

Spearbit: Fixed.

5.4 Gas Optimization

5.4.1 Pool addresses cannot be computed on-chain

Severity: Gas Optimization

Context: MarginalV1PoolDeployer.sol#L19

Description: The `MarginalV1PoolDeployer.deploy()` function is used by the factory contract to deploy new pool contracts (MarginalV1Factory.sol#L85-L90). The function uses the `CREATE2` opcode to generate the address of a new pool (MarginalV1PoolDeployer.sol#L16-L18). However, the function also passes constructor arguments to the `MarginalV1Pool` contract during deployment:

```
new MarginalV1Pool{
    salt: keccak256(
        abi.encode(msg.sender, token0, token1, maintenance, oracle)
    )
}(msg.sender, token0, token1, maintenance, oracle)
```

This makes it impossible to compute pool addresses on-chain because the `CREATE2` address computation scheme requires hashing the initialization bytecode of a contract (EIP-1014), which has constructor arguments appended to it. Since the initialization bytecode of a contract is not accessible from EVM, the hash cannot be computed on-chain dynamically for different sets of constructor arguments.

As a result, the periphery contracts will have to get pool addresses from the `MarginalV1Factory` contract in all user functions, which consumes more gas (calling a contract + reading a storage value from it) than computing addresses (data encoding + 2 `keccak256` invocations).

Recommendation: Consider passing `MarginalV1Pool` constructor arguments via a transient-like storage variable. For reference, consider using the implementation from Uniswap V3: UniswapV3PoolDeployer.sol#L20-L36.

Marginal: Agreed, but not sure how we do this without exceeding max contract bytecode size, which was the reason I resorted to this route of passing in constructor args.

Spearbit: Acknowledged.

5.4.2 Add short-circuiting in `stateSynced()` to save gas

Severity: Gas Optimization

Context: MarginalV1Pool.sol#L223-L234

Description: The `stateSynced()` function is used to update the `blockTimestamp` and `tickCumulative` state. Even when the state is already synced, it continues to calculate the new state. To save gas and avoid redundant calculations, it's possible to incorporate a short-circuiting in this logic.

Recommendation: It is recommended to insert a short-circuiting, similar to the code snippet below:

```
  function stateSynced() private view returns (State memory) {
    State memory _state = state;
+   if(_state.blockTimestamp == _blockTimestamp()) return _state;
    unchecked {
        _state.tickCumulative +=
          int56(_state.tick) *
          int56(uint56(_blockTimestamp() - _state.blockTimestamp)); // overflow desired
      _state.blockTimestamp = _blockTimestamp();
    }
    return _state;
  }
```

Marginal: Fixed in commit 4db2b05f.

Spearbit: Fixed.

5.4.3 Redundant check when transferring tokens out for position settlement

Severity: Gas Optimization

Context: MarginalV1Pool.sol#L509, MarginalV1Pool.sol#L550

Description: When opening a position, its size is validated to be non-zero. Hence, the amount to be transferred out (`position.size` + `position.margin` + `rewards`) will be non-zero.

Recommendation: The non-zero check can be removed.

```
- if (amount0 < 0)
  TransferHelper.safeTransfer(
    token0,
    recipient,
    uint256(-amount0)
  );

- if (amount1 < 0)
  TransferHelper.safeTransfer(
    token1,
    recipient,
    uint256(-amount1)
  );
```

Marginal: Agreed, but contract size seems to increase significantly with this change, so going to pass for now. If there's room left after resolving other issues, will return and implement.

Deployment Bytecode sizes before:

Contract	Bytecode size
MarginalV1PoolDeployer	- 24,466B (99.55%)
MarginalV1Pool	- 24,012B (97.70%)

Deployment Bytecode sizes after:

Contract	Bytecode size
MarginalV1PoolDeployer	- 24,554B (99.91%)
MarginalV1Pool	- 24,100B (98.06%)

Spearbit: Acknowledged.

5.4.4 Redundant `shares` check when removing liquidity

Severity: Gas Optimization

Context: MarginalV1Pool.sol#L873

Description: The referenced checks are redundant:

- If `shares == 0`, then `liquidityDelta = 0` => `amount0`, `amount1` will both be 0, resulting in no state change except for a `Burn()` event emission. Initially when `_totalSupply = 0`, `mulDiv` will revert.
- If `shares > _totalSupply`, it will revert with `InvalidLiquidityDelta()`, or in the worst case, revert with `ERC20: burn amount exceeds balance`.

Recommendation: Remove the referenced line.

```
- if (shares == 0 || shares > _totalSupply) revert InvalidShares();
```

Marginal: Fixed in commit edfbd6e5.

Spearbit: Fixed.

5.4.5 `initialized` is set but unused

Severity: Gas Optimization

Context: MarginalV1Pool.sol#L57, MarginalV1Pool.sol#L195

Description: `initialized` is set to `true` upon initialization, but is subsequently unused. For checking pool initialization status, it checks if the price set is non-zero.

```
if (state.sqrtPriceX96 > 0) revert Initialized();
```

Recommendation: Remove the `initialized` variable.

Marginal: Agreed `initialized` is not needed, but wanted to pack `State memory state` into 512 for gas reasons. Going to keep `initialized` in the state struct for the time being, unless other changes (like liquidation rewards revamp for instance) call for another state variable that we can replace `bool` with e.g. `uint8`.

Spearbit: Acknowledged.

5.4.6 `State` struct can be re-organized to reduce bytecode size

Severity: Gas Optimization

Context: MarginalV1Pool.sol#L49-L58

Description: The `State` structure takes up more storage slots than necessary. It can be re-organised to take up less storage slots such that the bytecode size is reduced.

Recommendation: A suggested `State` structure can be found below. While there is a better arrangement that takes up 1 less storage slot, it seems that it will cause the bytecode size to exceed the size limit.

```
struct State {
  uint128 liquidity;
  uint160 sqrtPriceX96;
  uint96 totalPositions; // > ~ 2e20 years at max per block to fill on mainnet
  int24 tick;
  uint32 blockTimestamp;
  int56 tickCumulative;
  uint8 feeProtocol;
  bool initialized;
}
```

Deployment Bytecode sizes before:

Contract	Deployment Bytecode size
MarginalV1PoolDeployer	- 24,577B (100.00%)
MarginalV1Pool	- 24,123B (98.15%)

Deployment Bytecode sizes after:

Contract	Deployment Bytecode size
MarginalV1PoolDeployer	- 24,511B (99.73%)
MarginalV1Pool	- 24,057B (97.88%)

Marginal: Fixed to two slots in commit 8258c799.

Just below contract size limits for tests to work:

Contract	Deployment Bytecode size
MarginalV1PoolDeployer	- 24,586B (100.04%)
MarginalV1Pool	- 24,132B (98.19%)

Saves ~5K gas (`open` was about 182K). See the `MarginalV1Pool` gas summary:

Method	Times called	Min.	Max.	Mean	Median
`__init__`	7	4557501	4557501	4557501	4557501
`adjust`	22	954	96962	58523	72866

Method	Times called	Min.	Max.	Mean	Median
approve	2	24376	24376	24376	24376
balanceOf	23	24248	24248	24248	24248
burn	12	886	97854	31479	15291
collectProtocol	2	10837	83825	47331	47331
factory	4	21886	21886	21886	21886
fee	32	21931	21931	21931	21931
fundingPeriod	4	21645	21645	21645	21645
initialize	3	3115	53679	36824	53679
liquidate	9	820	84149	23670	820
liquidityLocked	20	23656	23656	23656	23656
maintenance	89	21607	21607	21607	21607
mint	19	1077	143550	45991	29955
open	39	1547	177525	87149	60532
oracle	4	21688	21688	21688	21688
positions	53	30855	30855	30855	30855
protocolFees	12	23491	23491	23491	23491
reward	8	21381	21381	21381	21381
secondsAgo	4	21579	21579	21579	21579
setFeeProtocol	6	11239	14994	13136	13138
settle	17	1242	103172	57823	81478
state	183	26278	26278	26278	26278
swap	27	781	87398	40578	54764
tickCumulativeRateMax	1	21469	21469	21469	21469
token0	56	228	21292	11888	21292
token1	55	844	21908	12716	21908
totalSupply	20	23418	23418	23418	23418
transferFrom	1	31984	31984	31984	31984

Spearbit: Fixed.

5.4.7 Redundant zero value check on `protocolFees`

Severity: Gas Optimization

Context: MarginalV1Pool.sol#L916-L917

Description: The zero value check on `protocolFee` is redundant because (1) the function is permissioned, and (2) it'll revert later from sub-overflow should either value be zero. Furthermore, once fees start accruing, fee amounts are minimally 1, so this check would waste gas.

Recommendation: The check can be removed.

Marginal: Fixed in commit 5b10bca0.

Spearbit: Fixed.

5.4.8 `protocolFees` state can be directly set to `1` upon fee claims

Severity: Gas Optimization

Context: MarginalV1Pool.sol#L921, MarginalV1Pool.sol#L924

Description: When claiming protocol fees, the `protocolFees` struct variable slot isn't cleared for gas savings. The current implementation is gas inefficient because it does a subtraction on `protocolFees` when setting its new value.

Recommendation: `protocolFees` can be set directly to `1`.

```
- protocolFees.token0 -= amount0;
+ protocolFees.token0 = 1;

- protocolFees.token1 -= amount1;
+ protocolFees.token1 = 1;
```

Marginal: Fixed in commit a266cb05.

Spearbit: Fixed.

5.4.9 Redundant token checks for marginal pool deployment

Severity: Gas Optimization

Context: MarginalV1Factory.sol#L68

Description: Assuming only the canonical UniV3Factory is used, the checks

```
if (tokenA == tokenB || token0 == address(0)) revert InvalidTokens();
```

are redundant as UniswapV3Factory's `createPool()` does these checks. Hence, `getPool()` should return null address if either condition is satisfied, which will revert with the `InvalidOracle()` custom error.

Recommendation: The checks can be removed. For clarity, consider documenting the rationale for removing them.

Marginal: Fixed in commit d7017c09.

Spearbit: Fixed.

5.5 Informational

5.5.1 Arbitrage opportunities when opening, settling and liquidating positions

Severity: Informational

Context: MarginalV1Pool.sol

Description: When a position is opened, an internal swap happens from one pool token to the other. There is an explicit price impact for this trade which is proportional with the amount of liquidity locked by the position. Once a trader decides to settle their position, liquidity is returned to the pool and the debt is settled. The price impact of this action depends on the price action of `sqrtPriceX96` while the position was open.

It is important to note that an internal swap occurs during position settlement, but it does not happen when a position is liquidated. There are clear opportunities for the trader to back-run these key position actions by arbitraging between the marginal and univ3 pools.

Recommendation: It is unclear what solution there would be for this as it is inherent in the design of marginal pools. Ensure this is documented to traders and liquidity providers who may at times suffer significant short-term impermanent loss as compared to their univ3 pool counterpart.

Marginal: Agreed. Will document for users in docs.

Spearbit: Acknowledged.

5.5.2 JIT liquidity allows opening positions with lower debts and margin

Severity: Informational

Context: SqrtPriceMath.sol#L20, Position.sol#L144-L145

Description: As per the whitepaper, opening a position happens in two steps:

1. Some amount of liquidity is removed from the pool.
2. A portion of the liquidity is swapped on the updated pool's curve.
3. The bought tokens are added to the position size and the sold tokens are added to its debt.

Due to the underlying swapping, the result of opening a position depends on the amount of available liquidity in the pool and the desired size of the position. When there's more liquidity, the price impact of the internal swap is reduced, resulting in a better price of the swap. As a result, a position opened in a pool with higher liquidity will have a lower debt and a lower minimal margin than a position opened in a pool with lower liquidity (given the same position size).

Since newly provided liquidity is injected immediately (MarginalV1Pool.sol#L840), it's possible to manipulate a position's debt and minimal margin in one transaction. Consider this scenario:

1. A user takes a flash loan.
2. The user adds the loan as liquidity into a Marginal pool.
3. The user opens a position.
4. Since the added liquidity has reduced the price impact of swapping, the position has a lower debt and minimal margin than it'd had before the loaned amount was added.
5. The user removes liquidity and repays the loan.

Even though this scenario incurs costs on the user (e.g., paying the flash loan fee and the swap fees, swapping a portion of tokens back to compensate the internal swap, etc.) it gives the profit in terms of a reduced minimal margin, which can potentially be enough to pay the costs and still secure a profit.

Recommendation: We recommend monitoring such scenarios on-chain to better understand how and when they can happen and what are their impact. In case a mitigation is needed, delaying the injection of liquidity by at least one block can be a working solution.

Marginal: Agreed, will monitor -- believe Uniswap would also be susceptible to this type of tactic, so might be difficult in practice given costs to repay flash loan.

Spearbit: Acknowledged.

5.5.3 Changes to funding rate cap parameters should consider tick bounds

Severity: Informational

Context: MarginalV1.sol#L123, Position.sol#L356-L359

Description: `sqrt(P)` values are preferred over `P` in the oracle library to utilize the UniV3 library function `TickMath.getSqrtRatioAtTick()` when converting from averaged tick to price. This means that the computed `arithmeticMeanTick` = $\frac{2[(a_{t+\tau}-a_t)-(b_{t+\tau}-b_t)]}{T}$ is double its expected value, which possibly exceeds the tick boundaries allowed.

Because the funding rate is capped, the current configured parameters of `tickCumulativeRateMax = 920` and `fundingPeriod = 604800` ensures that the position would be very much liquidatable before the desired τ of `(MAX_TICK + 1) * fundingPeriod / 2 / tickCumulativeRateMax = 291642778` (~9.247 years) can be reached.

Recommendation: Changes to `tickCumulativeRateMax` or `fundingPeriod` should check that τ is of sufficiently high value to prevent `arithmeticMeanTick` from exceeding tick boundaries.

Marginal: Agreed. Documented in f7bf5a44.

Spearbit: Fixed.

5.5.4 Unused import

Severity: Informational

Context: MarginalV1Pool.sol#L26

Description: The import of `IMarginalV1PoolDeployer` is not used.

Recommendation: It is recommended to remove the import.

Marginal: Fixed in commit cd331c46.

Spearbit: Fixed.

5.5.5 Inconsistent solidity versions

Severity: Informational

Context: Global scope

Description: All files in-scope contain employ a floating pragma version, which is not recommended in production.

Recommendation: Consider using a fixed pragma version for each file.

Marginal: Disagree, given the thought is to make interfaces and libraries where the pragma is floating available to use for other projects that have `@marginal/v1-core as a dependency`. See for example `@marginal/v1-periphery`.

However, there is a problem with `IMarginalV1Pool.sol` pragma. It should be floating `^0.8.0` I think, given inherits from OZ's IERC20.sol.

Fixed in commit ddaad60f.

Spearbit: Agree with the project's response, the floating pragma can be used as it may serve as a dependency for other projects.

5.5.6 Unsolved `TODO`s

Severity: Informational

Context: MarginalV1Pool.sol, OracleLibrary.sol, Position.sol

Description: There are `TODO` comments in the files, most of which are questions related to this security review. However, it is recommended to remove all `TODO` comments before completing the review.

Recommendation: It's not recommended to include `TODO` in production contracts. They should be fixed or removed.

Marginal: Agreed. Will remove it before deployment once TODOs are resolved (mostly tests). Partially fixed in commit 98eb3e55.

Spearbit: Consider this issue fixed because the project will continue to implement the tests and remove the pending TODOs.

5.5.7 Duplicate fee calculation implementation across different libraries

Severity: Informational

Context: Position.sol#L227-L240, SwapMath.sol#L43-L49

Description: The referenced functions share identical implementations for fee calculations which are located in different libraries.

Recommendation: For better code maintainability, consider combining and/or abstracting the fee calculations to a separate library.

Marginal: Agreed, but slightly different now given fix to the issue "Exact output swaps pay less fees than exact input swaps" in commit 8e21eff8. So will pass on making changes for now.

Spearbit: Acknowledged.

5.5.8 `MarginalV1Pool` provides no way to check if a position can be liquidated

Severity: Informational

Context: MarginalV1Pool.sol#L630-L631

Description: The `MarginalV1Pool.liquidate()` function is used to liquidate underwater positions. To check if a position can be liquidated, the function calls `Position.sync()` and `Position.safe()` (MarginalV1Pool.sol#L622-L631). However, while being library functions, the two functions are not exposed in the `MarginalV1Pool` contract, and thus liquidators cannot check if a position can be liquidated before trying to liquidate it.

This makes it harder for liquidators to monitor positions that can be liquidated, since they have to simulate `MarginalV1Pool.liquidate()` transactions instead of making read-only calls to pools.

Recommendation: In `MarginalV1Pool`, consider implementing a public read-only function that returns `true` when a position can be liquidated and `false` otherwise.

Marginal: Agree, but I think this should be on the periphery since I'd guess most positions get opened through the periphery position manager NFT. Fixed in commit cb3e7aa8.

Spearbit: The `NonfungiblePositionManager.positions()` function was updated to return a `safe` value, which equals `true` when a position cannot be liquidated and `false` when a position can be liquidated. The value is obtained in the same way it's obtained in the `MarginalV1Pool.liquidate()` function. Fixed.