yAudit MarginZero Review

Review Resources:

- Code repository

Auditors:

- HHK
- Panda

Table of Contents

Review Summary

MarginZero

MarginZero provides an options trading platform built on top of Uniswap v3.

The contracts of the MarginZero Repo were reviewed over nine days. Two auditors performed the code review between October 21st and October 31st, 2024. The repository was under active development during the review, but the review was limited to the latest commit for the MarginZero repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
src
├── PositionManager.sol
├── apps
│   ├── options
│   │   ├── OptionMarketOTMFE.sol
│   │   └── pricing
│   │       ├── OptionPricingLinearV2.sol
│   │       └── fees
│   │           ├── ClammFeeStrategyV2.sol
├── handlers
│   ├── V3BaseHandler.sol
│   ├── hooks
│   │   └── BoundedTTLHook_0day.sol
│   └── uniswap-v3
│       ├── LiquidityManager.sol
│       └── UniswapV3Handler.sol
├── swapper
│   ├── OnSwapReceiver.sol
│   ├── OneInchSwapper.sol
│   ├── SwapRouterSwapper.sol
```

After the findings were presented to the MarginZero team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, MarginZero and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	The contracts implement proper access control mechanisms with owner-only functions for critical operations and whitelisting.
Mathematics	Good	The mathematical operations for options pricing and liquidity calculations are implemented correctly, considering precision and overflows properly. Some minor improvements could be made to the fee calculations.
Complexity	Good	The codebase follows established patterns from Uniswap V3 and maintains reasonable complexity. The options trading logic is well-structured and modular.
Libraries	Good	The contracts make good use of established libraries. External dependencies are minimal and well-vetted.
Decentralization	Low	The centralized settlement process could block users' funds. This creates a central point of failure that could impact users' ability to exit positions.
Code stability	Good	The codebase appears stable with good test coverage.
Documentation	Good	The code is well documented, with clear NatSpec comments explaining the functionality.
Monitoring	Average	While basic events are implemented, some key state changes lack events, making monitoring more difficult.
Testing and verification	Good	The contracts have comprehensive test coverage.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.

- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

1. High - `cache.sqrtPriceX96` is not set when creating sub-positions

Technical Details

In the function `_splitPosition()` the liquidity is burned and then added back into multiple sub positions allowing traders to find the best strike for them.

When depositing back into sub positions, the function will calculate the liquidity to be deposited using `LiquidityAmounts.getLiquidityForAmounts()`, which takes the current tick `sqrtPriceX96` as a parameter. Instead of querying the current price, the function uses `cache.sqrtPriceX96`, which is not set; this means the call will be made with a price of 0.

The underlying function `addLiquidity()` recalculates the liquidity to be added before calling `pool.mint()`.

The result is that the `_splitPosition()` will use a different amount of liquidity than was added. If the liquidity stored is less than what was added, the liquidity provider will lose some of his liquidity. If the liquidity stored is more than what was added, the liquidity provider and trader might run into an underflow when removing the whole position. If other liquidity providers have liquidity at the same ticks, one might withdraw more than he should at the cost of other liquidity providers.

Poc: Copy and paste in `UniswapV3Handler.t.sol` and run `forge test --mt testSplitPositionZero -vvv`

```
function testSplitPositionZero() public {
        TestVars memory vars;

        // Get current price and tick
        (vars.sqrtPriceX96, vars.currentTick,,,,,) = pool.slot0();
        int24 tickSpacing = pool.tickSpacing();

        // Calculate tick range that spans the current tick
        int24 initialTickLower = ((vars.currentTick / tickSpacing) * tickSpacing)
- 100 * tickSpacing;
        int24 initialTickUpper = ((vars.currentTick / tickSpacing) * tickSpacing)
+ 100 * tickSpacing;

        // Define split ticks
        int24[] memory splitTicks = new int24[](4);
        splitTicks[0] = initialTickLower;
        splitTicks[1] = initialTickLower + (initialTickUpper - initialTickLower)
/ 4;
        splitTicks[2] = initialTickLower + (initialTickUpper - initialTickLower)
/ 2;
        splitTicks[3] = initialTickUpper;

        // Set initial amounts
        uint256 amount0Desired = 10000 * 10 ** 6; // 10,000 USDC
        uint256 amount1Desired = 5 * 10 ** 18; // 5 ETH

        handler.registerHook(
            address(1),
            IHandler.HookPermInfo({
                onMint: false,
                onBurn: false,
                onUse: false,
                onUnuse: false,
                onDonate: false,
                allowSplit: true
            })
        );

        // Mint initial position
        vm.startPrank(owner);
        USDC.mint(owner, amount0Desired);
        ETH.mint(owner, amount1Desired);
        USDC.approve(address(positionManager), amount0Desired);
        ETH.approve(address(positionManager), amount1Desired);

        vars.liquidity = LiquidityAmounts.getLiquidityForAmounts(
            vars.sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(initialTickLower),
            TickMath.getSqrtRatioAtTick(initialTickUpper),
            amount0Desired,
            amount1Desired
        );

        V3BaseHandler.MintPositionParams memory mintParams =
V3BaseHandler.MintPositionParams({
            pool: IV3Pool(address(pool)),
            hook: address(1),
            tickLower: initialTickLower,
            tickUpper: initialTickUpper,
            liquidity: vars.liquidity
        });

        console.log("Initial liquidity at the 3 sub positions");
        for (uint256 i = 0; i < splitTicks.length - 1; i++) {
            (uint256 liquidity,,,,) =
pool.positions(keccak256(abi.encodePacked(address(handler), splitTicks[i],
splitTicks[i + 1])));
            console.log(liquidity);
        }

        positionManager.mintPosition(IHandler(address(handler)),
abi.encode(mintParams, ""));

        vars.tokenId =
            handler.getHandlerIdentifier(abi.encode(address(pool), address(1),
initialTickLower, initialTickUpper));

        {
            console.log("initial liquidity added");
            (uint128 liquidity,,,,,,,,,) = handler.tokenIds(vars.tokenId);
            console.log(liquidity);
            console.log("initial real liquidity added");
            (liquidity,,,,) =
pool.positions(keccak256(abi.encodePacked(address(handler), initialTickLower,
initialTickUpper)));
            console.log(liquidity);
        }

        // Split position
        V3BaseHandler.SplitPositionParams memory splitParams =
V3BaseHandler.SplitPositionParams({
            user: owner,
            pool: IV3Pool(address(pool)),
            hook: address(1),
            tickLower: initialTickLower,
            tickUpper: initialTickUpper,
            tickSplits: splitTicks
        });

        console.log("-------------wildcard--------------");

        bytes memory splitData =
abi.encode(V3BaseHandler.WildcardActions.SPLIT_POSITION,
abi.encode(splitParams));
        positionManager.wildcard(IHandler(address(handler)), splitData);

        // Check new positions
        for (uint256 i = 0; i < splitTicks.length - 1; i++) {
            uint256 currentTokenId =
                handler.getHandlerIdentifier(abi.encode(address(pool),
address(1), splitTicks[i], splitTicks[i + 1]));

            console.log("Liquidity of sub position handler think it has");
            console.log(handler.balanceOf(owner, currentTokenId));
            (uint256 liquidity,,,,) =
pool.positions(keccak256(abi.encodePacked(address(handler), splitTicks[i],
splitTicks[i + 1])));
            console.log("Actual liquidity of sub position");
            console.log(liquidity);
        }

        assertEq(handler.balanceOf(owner, vars.tokenId), 0, "Initial position
should be burned");

        vm.stopPrank();
    }
```

Impact

High.

Recommendation

Use the return value from the `addLiquidity()` function.

Developer Response

Fix - 1ca556da190c89bdb51a2837e5c6384adbc9b8cf and bc4bc1c99cfd801cb5d1874ee393b557ec234e77

Medium Findings

1. Medium - `_splitPosition()` will not add back all the liquidity (especially when the current price is in range)

Technical Details

In the function _splitPosition() the liquidity is burned and then added back into multiple sub positions allowing traders to find the best strike for them.

1 - When depositing back into sub positions, the function will compute liquidity per ticks for each asset: token A and token B. Using the full amount of liquidity removed and the total amount of ticks. When the price is in range, the liquidity removed will be a mix of 2 tokens. When adding back the liquidity, the ticks below the current price will only take one token, and those above will only take the other. However, the current implementation will try to add the same share of tokens A and B per sub positions instead of just adding the token needed, depending on whether we're above or below the price.

This will result in less liquidity added back; the tokens will be sent back to the liquidity provider. This can be annoying as it allows a malicious user to remove a part of the liquidity available, which will reduce the volume and the potential fees for users and the protocol.

Poc: Copy and paste in `UniswapV3Handler.t.sol` and run `forge test --mt testSplitPositionLess -vvv`

```
function testSplitPositionLessLiquidity() public {
        TestVars memory vars;

        // Get current price and tick
        (vars.sqrtPriceX96, vars.currentTick,,,,,) = pool.slot0();
        int24 tickSpacing = pool.tickSpacing();

        // Calculate tick range that spans the current tick
        int24 initialTickLower = ((vars.currentTick / tickSpacing) * tickSpacing)
- 100 * tickSpacing;
        int24 initialTickUpper = ((vars.currentTick / tickSpacing) * tickSpacing)
+ 100 * tickSpacing;

        // Define split ticks
        int24[] memory splitTicks = new int24[](4);
        splitTicks[0] = initialTickLower;
        splitTicks[1] = initialTickLower + (initialTickUpper - initialTickLower)
/ 4;
        splitTicks[2] = initialTickLower + (initialTickUpper - initialTickLower)
/ 2;
        splitTicks[3] = initialTickUpper;

        // Set initial amounts
        uint256 amount0Desired = 10000 * 10 ** 6; // 10,000 USDC
        uint256 amount1Desired = 5 * 10 ** 18; // 5 ETH

        handler.registerHook(
            address(1),
            IHandler.HookPermInfo({
                onMint: false,
                onBurn: false,
                onUse: false,
                onUnuse: false,
                onDonate: false,
                allowSplit: true
            })
        );

        // Mint initial position
        vm.startPrank(owner);
        USDC.mint(owner, amount0Desired);
        ETH.mint(owner, amount1Desired);
        USDC.approve(address(positionManager), amount0Desired);
        ETH.approve(address(positionManager), amount1Desired);

        vars.liquidity = LiquidityAmounts.getLiquidityForAmounts(
            vars.sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(initialTickLower),
            TickMath.getSqrtRatioAtTick(initialTickUpper),
            amount0Desired,
            amount1Desired
        );

        V3BaseHandler.MintPositionParams memory mintParams =
V3BaseHandler.MintPositionParams({
            pool: IV3Pool(address(pool)),
            hook: address(1),
            tickLower: initialTickLower,
            tickUpper: initialTickUpper,
            liquidity: vars.liquidity
        });

        uint256 prevEthBalance = ETH.balanceOf(owner);
        uint256 prevUsdcBalance = USDC.balanceOf(owner);

        positionManager.mintPosition(IHandler(address(handler)),
abi.encode(mintParams, ""));

        vars.tokenId =
            handler.getHandlerIdentifier(abi.encode(address(pool), address(1),
initialTickLower, initialTickUpper));

        // Split position
        V3BaseHandler.SplitPositionParams memory splitParams =
V3BaseHandler.SplitPositionParams({
            user: owner,
            pool: IV3Pool(address(pool)),
            hook: address(1),
            tickLower: initialTickLower,
            tickUpper: initialTickUpper,
            tickSplits: splitTicks
        });

        {
            uint256 addedEth = prevEthBalance - ETH.balanceOf(owner);
            console.log("Added ETH at initial liquidity");
            console.log(addedEth);
            uint256 addedUsdc = prevUsdcBalance - USDC.balanceOf(owner);
            console.log("Added USDC at initial liquidity");
            console.log(addedUsdc);
        }
        console.log("-------------wildcard-------------");

        bytes memory splitData =
abi.encode(V3BaseHandler.WildcardActions.SPLIT_POSITION,
abi.encode(splitParams));
        positionManager.wildcard(IHandler(address(handler)), splitData);

        assertEq(handler.balanceOf(owner, vars.tokenId), 0, "Initial position
should be burned");



        {
            uint256 addedEth = prevEthBalance - ETH.balanceOf(owner);
            console.log("Added ETH at split");
            console.log(addedEth);
            uint256 addedUsdc = prevUsdcBalance - USDC.balanceOf(owner);
            console.log("Added USDC at split");
            console.log(addedUsdc);
        }

        vm.stopPrank();
    }
```

2 - Additionally, even when out of range, the liquidity will not be added the same way as liquidity is not linear but slightly changes on each tick.

See this test (it can be copy pasted from UniswapV3Handler.t.sol then run forge test --mt testLiquiditySplit -vvv:

```
function testLiquiditySplit() public {
        //setup: tickCurrent: 200311, tickSpacing: 10,desiredAmount0:
100000000e6,desiredAmount1: 100 ether,desiredTickLower: 200010,desiredTickUpper:
201010,
        (uint160 sqrtPriceX96, int24 tickCurrent, , , , , ) = pool.slot0();
        (uint128 liquidityGross, int128 liquidityNet, , , , , ) =
pool.ticks(200010);//liquidity that was added in the setup

        (uint total0,uint total1) =
LiquidityAmounts.getAmountsForLiquidity(sqrtPriceX96,
TickMath.getSqrtRatioAtTick(200010), TickMath.getSqrtRatioAtTick(201010),
liquidityGross);
```

```
        console.log("token0 and token1 added in the setup:");
        console.log(total0);
        console.log(total1);

        uint totalTickAmount0;
        console.log("USDC per ticks:");
        for (int24 tick = 200320; tick < 201010; tick += 10) {
            (uint tickAmount) =
LiquidityAmounts.getAmount0ForLiquidity(TickMath.getSqrtRatioAtTick(tick),
TickMath.getSqrtRatioAtTick(tick + 10), liquidityGross);
            totalTickAmount0 += tickAmount;
            console.log(tickAmount);
        }
        uint totalTickAmount1;
        console.log("ETH per ticks:");
        for (int24 tick = 200010; tick < 200310; tick += 10) {
            (uint tickAmount) =
LiquidityAmounts.getAmount1ForLiquidity(TickMath.getSqrtRatioAtTick(tick),
TickMath.getSqrtRatioAtTick(tick + 10), liquidityGross);
            totalTickAmount1 += tickAmount;
            console.log(tickAmount);
        }

        (uint amount0InsideCurrentTick, uint amount1InsideCurrentTick) =
LiquidityAmounts.getAmountsForLiquidity(sqrtPriceX96,
TickMath.getSqrtRatioAtTick(200310), TickMath.getSqrtRatioAtTick(200320),
liquidityGross);

        console.log("total USDC in the range:");
        console.log(totalTickAmount0 + amount0InsideCurrentTick);
        console.log("total ETH in the range:");
        console.log(totalTickAmount1 + amount1InsideCurrentTick);

        assertApproxEqAbs(totalTickAmount0 + amount0InsideCurrentTick, total0,
100); //100wei max precision loss
        assertApproxEqAbs(totalTickAmount1 + amount1InsideCurrentTick, total1,
100); //100wei max precision loss
    }
```

We notice that the USDC and ETH per tick are not equal around the whole range but slightly change for every tick. Because the price changes at every tick, the token amount can't be the same inside the whole range. It has to change as well.

image

Because of that, if we calculate the amountPerTick = total / totalTicks, it will not result in the same allocation since the amount per tick is not the same initially.

Impact

Medium. The new position's liquidity and concentration might differ from the initial position.

Recommendation

1 - Each sub position should check if it's above or below the current tick and add the corresponding token amount back.
2 - Could loop inside the initial range, and for each sub-position, get the sum for each tick inside that sub-position range.

Developer Response

Fixed in:

- 62ac769cce2dba7a3ae1c52f4dc8e896b484063c
- 7e3fcaefdf9054e7de0b11d25afd20cba6478c49
- 84f395cfdd181477859129696b56b127c9284ac8

Acknowledged that the new sub-position's liquidity might slightly differ from the original position.

2. Medium - inconsistent `_feeCalculation()` may lead to inaccurate fees stored

The `V3BaseHandler` stores the fees received by the pools. However, in some casese, the fees are not computed or at the wrong moment inside the functions, which can lead to inaccuracy.

Technical Details

The `V3BaseHandler` is forked from the Uniswap's `NonfungiblePositionManager` contract. It will hold Univ3 positions created by the users that option traders do not currently borrow.

Both use the same mechanism to compute and store the fees of each pool whenever there is an update to that pool. The `NonfungiblePositionManager` does it inside the function while the `V3BaseHandler` will call the internal function `_feeCalculation()`.

In the `NonfungiblePositionManager`, we can observe that the fees are always updated before the new liquidity is updated. This is important as the fees the position receives depend on its liquidity.
However, in the `V3BaseHandler,` there are two functions that don't implement this correctly:

- The function `_splitPosition()` doesn't call `_feeCalculation()` after burning the position.
- The function `mintPositionHandler()` calls the `_feeCalculation()` after updating the liquidity instead of calling it before.

This can lead to the wrong amount of fees stored which can lead to a revert of the `_collectFees()` function as the contract will try to claim more fees than it has.

Impact

Medium.

Recommendation

Add a `_feeCalculation()` call inside `_splitPosition()` and update the liquidity after calling `_feeCalculation()` inside `mintPositionHandler()`.

Developer Response

Fix - 8d9b9bf3a51fee0109ccaa0798ff7a891745e0e0.

Low Findings

1. Low - Hooks can block traders from settling their options

Technical Details

When adding liquidity, the liquidity providers can input a hook contract. A hook has to be registered before that. Anyone can do it by calling the function `registerHook()`.

The hook will then be called on every interaction with the liquidity. For example, when a trader tries to settle his option to make profits, the hook could revert on purpose.

When a trader buys an option, he can know if a hook is tied to the liquidity he is borrowing from. However, it might not be clear on the front end that a hook is tied and that this hook may block him from settling his option later. This creates a risk for non-technical users who might not check the hook contract or are unaware that a hook is tied to the liquidity they are using.

Impact

Low.

Recommendation

Consider only displaying liquidity of whitelisted hooks on the frontend or add an `onlyOwner` to the `registerHook()` function.

Developer Response

Acknowledged. The buyers should be aware of the hooks they are using.

2. Low - Missing slippage checks when adding and removing liquidity

Technical Details

The function `mintPosition()` and `burnPosition()` don't have a slippage check.

When adding liquidity if the prices change between the transaction broadcasting and execution, the amount of token A or B pulled from or sent to the user might differ from what the user initially expected.

Impact

Low.

Recommendation

Similar to the original `NonfungiblePositionManager.sol` from Uniswap, add `amount0Min` and `amount1Min` to the `_mintPositionData` and `_burnPositionData` parameters and check them against the amount returned by the pool.

Developer Response

Acknowledged. This can be done by router outside core, can be also governed by strict approvals.

3. Low - Unsafe use of `transferFrom` and `approve`

`safeTransferFrom` and `safeApprove` should be used across the codebase. The code will run on L2, reducing the risk with inconsistent ERC20 tokens. If it was to be deployed on Mainnet, using `safeFunction` instead of `function` is a must.

Technical Details

On Mainnet, some ERC20s, like USDT, don't return a boolean. The operation will always fail when using `transferFrom()` or `approve()`.

Impact

Low.

Recommendation

Change the transfer operations to use `safeTransferFrom`.

Developer Response

Acknowledged.

4. Low - `computeAddress()` will not return the correct pool on ZKsync

Technical Details

The function `computeAddress()` forked from the original `NonfungiblePositionManager()` will not work on ZKsync as the address computation for `CREATE` and `CREATE2` is different. This will make the transaction revert when trying to add liquidity.

Impact

Low.

Recommendation

Update the function when deploying on ZKsync, you can find the Uniswap's ZKsync version here.

Developer Response

Acknowledged. We will be careful, when deploying on zksync.

5. Low - Missing sweep functions

The swapper contracts are missing a sweep function.

Technical Details

The OnSwapReceiver contract is owned without any `onlyOwner` modifier, it appears the contract is missing a sweep function. A 'sweep' function might also benefit the other two swapper contracts.

Impact

Low

Recommendation

Add the sweep function when necessary.

Developer Response

Acknowledged, this will be added later.

6. Low - Option's positions can be split after expiry

Technical Details

The function `positionSplitter()` allows to split positions of an option into another option.

However, this can be called even after the option reaches expiry. This could allow a malicious user to move the positions of an option towards another so the `settleOption()` calls keep failing and users don't get their liquidity back.

Impact

Low.

Recommendation

Revert if the option expires.

Developer Response

Fix - e0e6e9dbe281314cbb108c19d49011f106220946.

7. Low - Incomplete slippage check in `mintOption()`

Technical Details

The function `mintOption()` allows to mint option using liquidity for liquidity providers.

The function implements a check on `params.maxCostAllowance` to make sure the user doesn't pay more fees than expected. But it doesn't allow the user to specify a max or min tick to revert if the price moved.
The premium will change depending on the current price of the `primePool`, so this should help to protect the user against high price variation, but this is incomplete.
Additionally, there is no expiry/max timestamp for the transaction apart from the `expiry` of the option itself.

Impact

Low.

Recommendation

Consider:

- Adding an expiry timestamp parameter at which the transaction should revert if it's executed after.
- A tick range parameter should be added that the transaction should revert if `primePool` is outside that range.

Developer Response

Acknowledged. This can be handled by the router outside the core. As, it shouldn't cause any issues rather than fails, I'm okay to keep it the way it is.

8. Low - No minimum size for options

Technical Details

Options can be minted with as much as one wei of liquidity in the function `mintOption()` as long as the premium is greater than 0.

While the gas cost might be a blocker for malicious users to spam small options, it is possible to mint an option with more liquidity and then call the function `positionSplitter()` to split this option into multiple smaller options. This would make it hard and expensive for the settler bots to settle all of them and might result in some users liquidity getting stuck for some time.

Impact

Low.

Recommendation

Consider not allowing splitting options below a certain threshold.

Developer Response

Acknowledged

Gas Saving Findings

1. Gas - Functions with unused parameters

Some arguments are never used.

Technical Details

```
File: apps/options/pricing/OptionPricingLinearV2.sol

103: function getOptionPrice(address hook, bool isPut, uint256 expiry, uint256
ttl, uint256 strike, uint256 lastPrice)

135: function getOptionPriceViaTTL(address hook, bool isPut, uint256 ttl, uint256
strike, uint256 lastPrice)
```

`hook` is never used for either of these functions.

OptionPricingLinearV2.sol#L103-L135

Impact

Gas savings.

Recommendation

If this is intentional, consider removing or commenting on these arguments from the function. Otherwise, implement the missing logic accordingly.

Developer Response

This is intentional, as we will be using the hook for custom pricing in the future :).

2. Gas - Pass on `tokensToPullFor...()` results to the next call

Technical Details

When calling `mintPosition()` and `unusePosition()` the functions will first call `tokensToPullForMint()` and `tokensToPullForUnUse()` which compute the amount of token 0 and 1 needed to add liquidity.

Then, the functions transfer them from the `msg.sender` and approve the `V3BaseHandler`. Finally, it calls the subsequent function on the `V3BaseHandler`: `mintPositionHandler()` and `unusePositionHandler()`. Inside these functions, the amount of token 0 and 1 will be recomputed even though they were already calculated, leading to extra gas usage.

It would be simpler and cheaper to pass them on from the `PositionManager` to the `V3BaseHandler` as function parameters.

Impact

Gas.

Recommendation

Pass the amount 0 and 1 from the `PositionManager` to the `V3BaseHandler` as function parameters for `mintPosition()` and `unusePosition()`.

Developer Response

Good suggestion, but would need to much refactor at this point and maybe not that much worth it atm.

3. Gas - Useless liquidity calculation in `_splitPosition()`

Technical Details

In the function `_splitPosition()` the `loopCache.newLiquidity` is computed using `LiquidityAmounts.getLiquidityForAmounts()` and then it adds liquidity using `_addLiquidity()`.

Since the function `_addLiquidity()` will recompute the liquidity to be added and return it, it's unnecessary to call `LiquidityAmounts.getLiquidityForAmounts()` before it.

Impact

Gas.

Recommendation

Remove the `LiquidityAmounts.getLiquidityForAmounts()` call and use the return value from `_addLiquidity()`.

Developer Response

Fix - bc4bc1c99cfd801cb5d1874ee393b557ec234e77.

4. Gas - Unused variable in `mintOption()`

Technical Details

The function `mintOption()` calculates the liquidity to donate and stores it in a memory variable `liquidityToDonate` but never use it.

Impact

Gas.

Recommendation

Remove the calculation and variable.

Developer Response

Fix - c43981e53d99a6553199e5be9ca04a04f5691587.

5. Gas - Function state mutability can be restricted to view

The function doesn't modify any state.

Technical Details

```
File: BoundedTTLHook_0day.sol

50: function onPositionUseBefore(bytes calldata _data) external {
```

BoundedTTLHook_0day.sol#L50

Impact

Gas savings.

Recommendation

Add the `view` function modifier.

Developer Response

There can be operations inside this that might change the state variable in the hook contract, so we can't keep it as view.

6. Gas - Structs can be packed into fewer storage slots

Each slot saved can avoid an extra Gsset (**20000 gas**) for the first setting of the struct. Subsequent reads, as well as write,s have smaller gas saving.s

Technical Details

```
File: OptionMarketOTMFE.sol

// @audit: 1 slot could be saved, by using a different order:
\*
 * uint256 opTickArrayLen; // (256 bits)
 * uint256 expiry; // (256 bits)
 * int24 tickLower; // (24 bits)
 * int24 tickUpper; // (24 bits)
 * bool isCall; // (8 bits)
 */

struct OptionData {
        uint256 opTickArrayLen;
        int24 tickLower;
        int24 tickUpper;
        uint256 expiry;
        bool isCall;
    }

// @audit: 1 slot could be saved, by using a different order:
\*
 * struct OptionMarketOTMFE.OptionTicks[] optionTicks; // (256 bits)
 * uint256 ttl; // (256 bits)
 * uint256 maxCostAllowance; // (256 bits)
 * int24 tickLower; // (24 bits)
 * int24 tickUpper; // (24 bits)
 * bool isCall; // (8 bits)
 */

struct OptionParams {
        OptionTicks[] optionTicks;
52:         int24 tickLower;
53:         int24 tickUpper;
54:         uint256 ttl;
55:         bool isCall;
56:         uint256 maxCostAllowance;
57:     }

// @audit: 1 slot could be saved, by using a different order:
\*
 * uint256 totalProfit; // (256 bits)
 * uint256 totalAssetRelocked; // (256 bits)
 * contract ERC20 assetToUse; // (160 bits)
 * bool isSettle; // (8 bits)
 * contract ERC20 assetToGet; // (160 bits)
 */

354: struct AssetsCache {
355:         ERC20 assetToUse;
356:         ERC20 assetToGet;
357:         uint256 totalProfit;
358:         uint256 totalAssetRelocked;
359:         bool isSettle;
360:     }
```

OptionMarketOTMFE.sol#L31, OptionMarketOTMFE.sol#L50, OptionMarketOTMFE.sol#L354

```
File: V3BaseHandler.sol

// @audit: 1 slot could be saved, by using a different order:
\*
 * uint256 userAmount0; // (256 bits)
 * uint256 userAmount1; // (256 bits)
 * uint256 totalTicks; // (256 bits)
 * uint256 amount0PerTick; // (256 bits)
 * uint256 amount1PerTick; // (256 bits)
 * uint160 sqrtPriceX96; // (160 bits)
 * int24 tickSpacing; // (24 bits)
 */

475: struct SplitPositionCache {
476:         uint160 sqrtPriceX96;
477:         uint256 userAmount0;
478:         uint256 userAmount1;
479:         int24 tickSpacing;
480:         uint256 totalTicks;
481:         uint256 amount0PerTick;
482:         uint256 amount1PerTick;
483:     }
```

V3BaseHandler.sol#L475

Impact

Gas savings.

Recommendation

Reorder the structures.

Developer Response

Fix - 945d1f4e639373beaa3ecfdd9201ae89eb2b0c36.

7. Gas - State variables only set in the constructor should be declared `immutable`

Technical Details

```
File: OptionMarketOTMFE.sol

191: START_TIME = _START_TIME;
```

OptionMarketOTMFE.sol#L191

Impact

Gas savings.

Recommendation

Use the `immutable` variable modifier.

Developer Response

Good suggestion, but this increases the deployment size, so will pass on this.

Informational Findings

1. Informational - Missing variables in natspec

Technical Details

- The function `getOptionPrice()` is missing the `ttl` parameter in the natspec.
- The function `_getPremiumAmount()` is missing the `ttl` parameter in the natspec.

Impact

Informational.

Recommendation

Add missing natspec.

Developer Response

Acknowledged.

2. Informational - Simplify `_addLiquidity()`

Technical Details

The function `_addLiquidity()` can be called with `self` as `true` or `false`. When set to `true`, the function will make the contract call itself to modify the `msg.sender`.

To reduce the code quantity and make this simpler, consider making the `addLiquidity()` function internal since there is no use for a normal user to call it and add a `sender` parameter.

When the `_addLiquidity()` is called with `self == true` then input `address(this)` as the sender.

Impact

Informational.

Recommendation

Apply the proposed changes.

Developer Response

Acknowledged.

3. Informational - `isAmount0` can change

Technical Details

In the function [`mintOption()`]() when removing liquidity, the function check if the amount removed is token 0 or 1 for each position and then update the `isAmount0` variable.

The issue is that the `isAmount0` variable is a boolean and not an array. This means that for every position in the option, the value might get updated to `true` or `false`.

Later on, this variable is used when distributing the premium fee. It will use the latest value and not the value found for each position since it's just a boolean.
This won't cause any issue in the current implementation as the tokens 0 and 1 are just distributed directly on their own. However this is not ideal and may result in errors if the code is updated later.

Impact

Informational.

Recommendation

Consider either making `isAmount0` an array and setting it for each position or if all pools are supposed to have the same token order (e.g., all Uniswap forks), then just determine `isAmount0` once using the `primePool` and not for every position.

Developer Response

It's require for us to check, as we want to verify the return, and make sure it matches with the assetToUse. It's gonna be same for all the UniswapV3Pool, we can definitely cache it.

4. Informational - Make sure the array length matches

Technical Details

The array `_ttlIV` might not match the length of the `_ttls` array. Make sure both lengths match before iterating.

```
File: OptionPricingLinearV2.sol

64: ttlToVol[_ttls[i]] = _ttlIV[i];
```

[OptionPricingLinearV2.sol#L64]()

Impact

Informational

Recommendation

Add a check before iterating over the arrays.

Developer Response

Fix - 750ba3f770f3fe6fcad9f9802920fc9ded45cd82.

5. Informational - Function state mutability can be restricted to pure

Technical Details

```
File: OptionMarketOTMFE.sol

198: function name() public view override returns (string memory) {

204: function symbol() public view override returns (string memory) {
```

OptionMarketOTMFE.sol#L198, src/apps/options/OptionMarketOTMFE.sol#L204

```
File: V3BaseHandler.sol

675: function tokensToPullForWildcard(bytes calldata _data) external view returns
(address[] memory, uint256[] memory) {
```

V3BaseHandler.sol#L675

Impact

Informational

Recommendation

Use the `pure` modifier.

Developer Response

Good suggestion, but there might be application using the state read, so better to keep it view to make it more generalized.

6. Informational - Missing event for state variable change

It’s recommended to have events when variables that reconfigure the protocol the owner sets are changed.

Technical Details

```
File: OptionPricingLinearV2.sol

52: function updateIVSetter(address _setter, bool _status) external onlyOwner {
53:         ivSetter[_setter] = _status;
54:     }

71: function updateVolatilityOffset(uint256 _volatilityOffset) external onlyOwner
returns (bool) {
72:         volatilityOffset = _volatilityOffset;
73:
74:         return true;
75:     }

80: function updateVolatilityMultiplier(uint256 _volatilityMultiplier) external
onlyOwner returns (bool) {
81:         volatilityMultiplier = _volatilityMultiplier;
82:
83:         return true;
84:     }

89: function updateMinOptionPricePercentage(uint256 _minOptionPricePercentage)
external onlyOwner returns (bool) {
90:         minOptionPricePercentage = _minOptionPricePercentage;
91:
92:         return true;
93:     }
```

OptionPricingLinearV2.sol#L52, OptionPricingLinearV2.sol#L71, OptionPricingLinearV2.sol#L80, OptionPricingLinearV2.sol#L89

```
File: V3BaseHandler.sol

811: function updateHandlerSettings(
812:         address _app,
813:         bool _status,
814:         address _hook,
815:         uint64 _newReserveCooldown,
816:         address _newFeeReceiver
817:     ) external onlyOwner {
818:         if (_app != address(0)) {
819:             whitelistedApps[_app] = _status;
820:         }
821:
822:         reserveCooldownHook[_hook] = _newReserveCooldown;
823:
824:         if (_newFeeReceiver != address(0)) {
825:             feeReceiver = _newFeeReceiver;
826:         }
827:     }
```

[V3BaseHandler.sol#L811]

```
File: BoundedTTLHook_0day.sol

77: function updateWhitelistedAppsStatus(address app, bool status) external
onlyOwner {
78:         whitelistedApps[app] = status;
79:     }
```

[BoundedTTLHook_0day.sol#L77]

Impact

Informational

Recommendation

Add events.

Developer Response

Good suggestion, but we did it to save storage space, we track these off-chain. So, will pass on this.

7. Informational - Unused import

The identifier is imported but never used within the file.

Technical Details

```
File: OptionPricingLinearV2.sol

7: import {ABDKMathQuad} from "./external/ABDKMathQuad.sol";
```

OptionPricingLinearV2.sol#L7

```
File: LiquidityManager.sol

5: import {IUniswapV3Factory} from "@uniswap/v3-
core/contracts/interfaces/IUniswapV3Factory.sol";
```

LiquidityManager.sol#L5

Impact

Informational

Recommendation

Remove the not-used import.

Developer Response

Fix - a64d1b0f6995e9a3e867a09d4d50107475b44946.

Final remarks

The MarginZero protocol demonstrates a well-engineered approach to options trading on Uniswap V3. However, the audit revealed several high- and medium-severity issues related to liquidity management, fee calculations, and position splitting that must be addressed before deployment. The codebase shows strong engineering practices with modular design, comprehensive testing, and thorough documentation. The identified vulnerabilities could impact core protocol functionality and user funds if not addressed.