BAIL security

Stryke / MarginZero
CLAMM Options

FINAL REPORT

May ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Audit - Stryke / MarginZero - CLAMM Options
Website	Stryke.xyz
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/marginzero-xyz/contracts/tree/84e2ea5ee97fa1dfe9070611bc764799fa6483f7
Resolution 1	https://github.com/marginzero-xyz/contracts/tree/e3c913addb54784d39cbec03691ff0a2051eba63

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution	Open
High	13	6		6	1	
Medium	17	10		6		1
Low	33	7	1	25		
Informational	8			8		
Governance	3			3		
Total	74	23	1	48	1	1

a) Bailsec recommends a follow-up audit due to the volume of H/M issues
b) Some high-risk issues are acknowledged because some contracts are not used in production or safety-checks are outsourced to a router

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

OptionMarketOTMFE

Disclaimer: IOptionPricingV2 is out of scope. The premium calculation has not been audited by Bailsec

The OptionMarketOTMFE contract allows users to buy and execute options which are based on UniswapV3 liquidity. This contract is deeply interconnected with the Shadow Handler as it basically uses the liquidity which is sitting in the handler.

Users can buy an option via the mintOption function which then removes single sided liquidity from the Handler contract for a specific position while the user pays a premium which is then distributed to the liquidity provider. It is possible to mint up to 20 sub-options under the same optionId which basically allows for aggregating liquidity from different protocols. However - **this protocol is only audited with the assumption of the Shadow handler being used**.

Once the option has reached expiry time or the strike price has been reached, it can be executed via the settleOption function which simply re-adds liquidity to the handler contract and optionally pays out a profit to the option owner.

Additionally the contract exposes the functionality of splitting sub-options into a new optionId.

It is important to mention that the protocol only works with pairs which are USD denominated, ie. ETH/USDT, while the USD asset will always be the putAsset.

Appendix: TTL

The expiry is calculated as follows:

a) Compute the remainder of the current 24 hour cycle:

> (block.timestamp−ttlStartTime[86400])mod86400

b) Subtract the remainder to calculate how many seconds remain:

> 86400–[(block.timestamp–ttlStartTime[86400])mod86400]

c) Adding this to the current time will yield the expiry

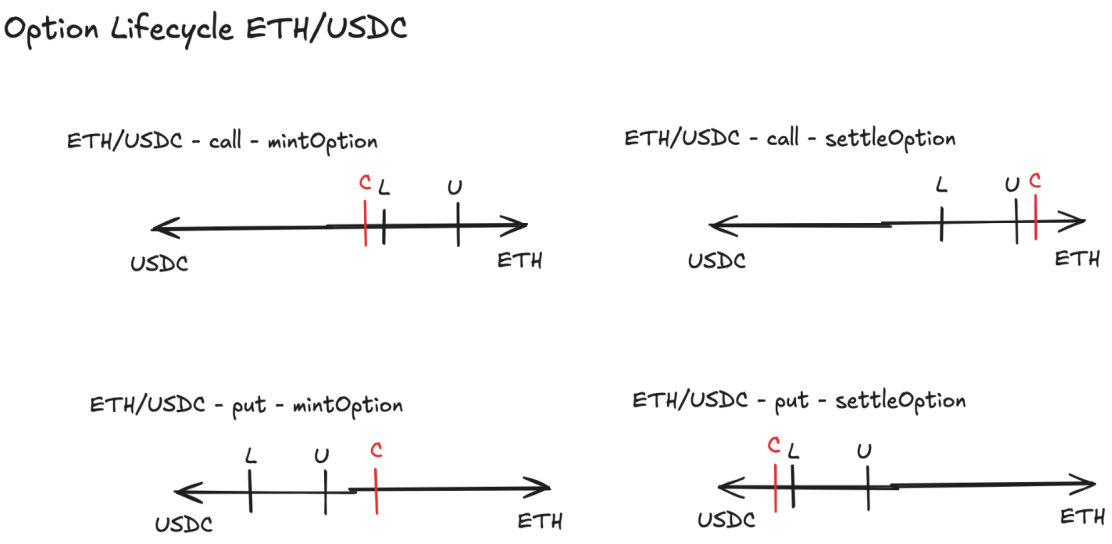
Appendix: Option Minting

Example:
> ETH/USDC
> Call option

Minting a call option is possible via the mintOption function while the liquidity is fully single sided as ETH (the price is below lowerTick). The desired liquidity will then be removed from the Handler and transferred as ETH to the Options contract, while the option minter must pay a protocol fee and a premium in ETH.

At this point, the option is considered fully OTM. If the price increases and ETH gets more valuable, the option starts to move ITM while being considered as fully ITM if the price moves above the upper tick. It can then be settled by the option owner or his corresponding delegate.

Minting a put option follows the exact same principles but in the other direction.

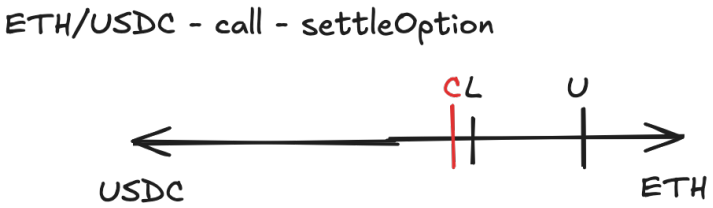


Example:
> ETH/USDC
> Call option

Settling an option is possible via the settleOption function. This function has many different callpaths based on the timing of the execution and the current price. Below we will explain all different scenarios.

Scenario 1: OTM and expired

If the price did not sufficiently increase after a CALL option has been minted, the option stays OTM which basically means the required liquidity to re-add consists of the exact same amount of ETH which has been received during the time of the option minting. The ETH amount will be simply re-added as if nothing has happened. The option is now considered as closed and the provider can withdraw his liquidity via the handler.



Scenario 2: ITM and non-expired

If the price did sufficiently increase after a CALL option has been minted and the strike price has been reached, an option is now considered ITM. The liquidity which has been withdrawn would now require only the PUT asset to be added while at the same time the CALL asset has increased significantly in value. The option creator now swaps (under the hood) all ETH to USDC which results in a significant leftover of USDC. The required USDC amount is then re-added as liquidity and the option owner gains the profit of the excess USDC which is left. The profit is basically the difference between the current price of ETH and the average curve price of the provided range.

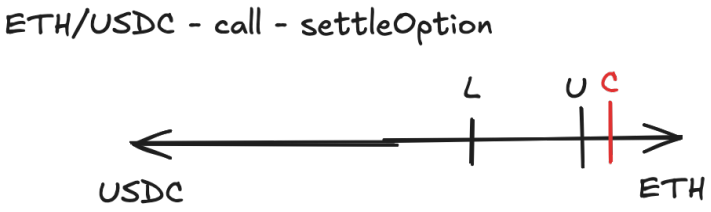
This can be illustrated with the following numerical example:

Let's assume a call option was minted by Alice for 2 ETH when it was trading at $2000 with a strike price of $2500. We will use a position of a single tick to make the example easier:
Now, ETH is trading at $3000 and Alice wants to exercise her call option. She calls the settleOption, under the hood, it will swap 2 ETH to $6000. $5000 will be re-added to the position that was used during the option creation, and Alice will receive $1000 in profits.

Which is the correct amount as:

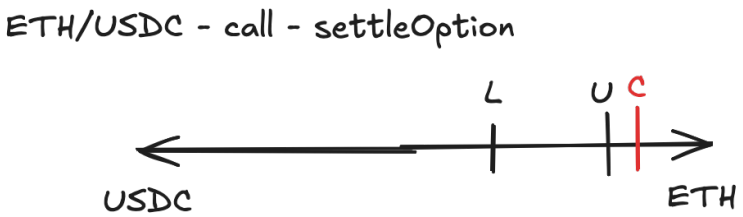
> (currentPrice - strikePrice) * size = (2500 - 3000) * 2 = $1000

If the position was on a wide range of ticks, the amount of USDC to deposit would have been lower, and Alice would have received more profits.



Scenario 3: ITM and expired

This scenario follows the exact same callpath as Scenario 2. The only difference is that only allowed settlers can execute this scenario.



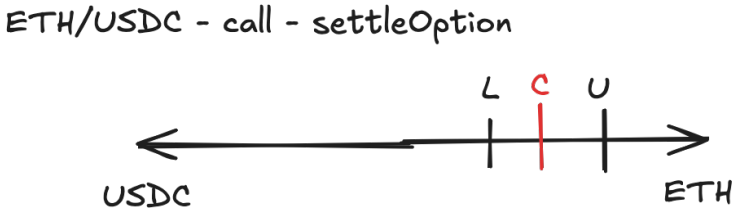
Scenario 4: Partially ITM and expired

If the ETH price has been increased and crossed lowerTick but not crossed upperTick (strike price not reached), a position is not considered ITM but has already gained partial profit due to the fact that ETH became worth more and the position now requires USD. If the liquidity would not have been removed, the ETH would have been swapped for USDC at the average curve price until the current price. Since this however has not happened, it would now theoretically be possible to swap the ETH to USDC using the current price which results in receiving more USDC than what is required to re-add liquidity. Instead of doing this explicit swap, the function just requires the caller to transfer in USDC while transferring out the excess ETH. The executor now gained a nominal profit because the ETH which was transferred out is worth more than USDC which was transferred in.

This can be illustrated with the following numerical example:

Let's assume a call option was minted by Alice for 2 ETH when it was trading at $2000 with a strike price of $2500. The position goes from $2000 to $2500.
ETH is trading at $2250 when the option expires, Alice can't exercise her option as it's not over her strike price.

The settler now has to call settleOption with enough token approved to re-deposit the liquidity. In this example, the position should be composed of 2304 USDC and 0.913 ETH. The settler must provide the 2304 USDC and will receive 1.087 ETH. The settler is in profit as he swaps the ETH at current price, he will receive more USDC than deposited: 1.087 * 2250 = $2445.75.



Appendix: Premium Calculation

Whenever an option is minted, the caller must provide a premium in the CALL or PUT asset. This necessary amount is determined based on the totalAssetWithdrawn value which is the aggregate of all sub-options. It is simply multiplied with the return value of optionPricing.getOptionPrice, which is out of scope. Below are example values which showcase the behavior of the calculation:

Core assumption: getOptionPrice returns price in current pair denomination

- ETH = 18 decimals
- USDC = 6 decimals
- ETH/USDC = 2500e-12
- USDC/ETH = 0.0004e12
- option prices are 90% of the real prices (for simplicity)

############## CALL execution #################

SCENARIO 1: ETH/USDC; token0 = CALL

- price = 2500e-12
- getOptionPrice = 2250e-12
- premiumInQuote = withdrawAmount * getOptionPrice / callAssetDecimals
- amount = premiumInQuote * callAssetDecimals / lastPrice
- premiumInQuote = 1e18 * 2250e-12 / 1e18
- premiumInQuote = 2250e-12 (2250 USDC)
- amount = 2250e-12 * 1e18 / 2500e-12
- amount = 0.9e18 (0.9 ETH)

SCENARIO 2: USDC/ETH; token1 = CALL

- price = 2500e-12
- getOptionPrice = 2250e-12
- premiumInQuote = withdrawAmount * getOptionPrice / callAssetDecimals
- amount = premiumInQuote * callAssetDecimals / lastPrice
- premiumInQuote = 1e18 * 2250e-12 / 1e18
- premiumInQuote = 2250e-12
- amount = 2250e-12 * 1e18 / 2500e-12
- amount = 0.9e18 (0.9 ETH)

############## PUT execution #################

SCENARIO 3: ETH/USDC; token1 = PUT

- price = 2500e-12
- getOptionPrice = 2250e-12
- withdrawAmount = totalAssetsWithdrawn * putAssetDecimals / strike
- premiumInQuote = withdrawAmount * getOptionPrice / putAssetDecimals
- withdrawAmount = 1250e6 * 1e6 / 3500e-12
- withdrawAmount = 3.57e23
- premiumInQuote = 3.57e23 * 2250e-12 / 1e6
- premiumInQuote = 803.25e6 (803.25 USDC)

SCENARIO 4: USDC/ETH; token0 = PUT

- price = 2500e-12
- getOptionPrice = 2250e-12
- withdrawAmount = totalAssetsWithdrawn * putAssetDecimals / strike
- premiumInQuote = withdrawAmount * getOptionPrice / putAssetDecimals
- withdrawAmount = 1250e6 * 1e6 / 1428e-12
- withdrawAmount = 8.75e23
- premiumInQuote = 8.75e23 * 2250e-12 / 1e6
- premiumInQuote = 1968e6 (1968 USDC)

Appendix: Option Profit

As already explained within “Appendix: Open Settlement”, an option is considered as profitable not only when ITM but also when the current price has already crossed the lowerTick boundary (for CALL; ETH/USDC).

The profit lies in the fact that the price shift changes the ratio of ETH/USDC for the same liquidity but ETH has already been withdrawn. The withdrawn ETH amount can then be swapped to USDC using the actual current price, which brings the user as nominal profit because it is not swapped over the average curve price over the provided range.

For further details we recommend reading the example which was provided in the corresponding appendix.

Appendix: Execution via delegator

The contract not only allows execution by the option owner but also via a delegatee. Any address can determine one or multiple delegatee's via the updateExerciseDelegate function. This address can then settle options on behalf of the owner while collecting the profit.

Appendix: Option Aggregation

Within mintOption, one tokenId is minted which is corresponding to the current optionId. Within this optionId there can be up to 20 sub-options which can correspond to different ranges, pools and hooks. Whenever this option is then settled, it will settle all sub-options with their desired liquidity.

Core Invariants:

INV 1: mintOption can only be called if a position is single sided

INV 2: A CALL option must have strike price above the current price

INV 3: A PUT option must have strike price below the current price

INV 4: Only the owner of an option can split it

INV 5: An expired option cannot be split

INV 6: A sub-option can be split with the maximum of its assigned liquidity

INV 7: _getPrice must always return the price for CALL asset in PUT asset

INV 8: After an option has expired, only whitelisted settlers can execute it

INV 9: Before an option has expired, only the owner or delegatee can execute it

INV 10: One optionId can only have up to 20 sub-options

INV 11: One optionId can have sub-options from different pools

INV 12: If a position is used, single-sided liquidity must always be sitting in the Option contract

INV 13: Within option settlement, ac.totalProfit must only be set if option is ITM

INV 14: Settling during in-range requires settler to provide corresponding amount0/1 while receiving amount0/1

INV 15: Within settleOption, amountToSwap must always correspond to the amount which was received during usePosition under the premise that liquidity is equivalent

INV 16: Profit during option settlement is always paid out in assetToGet

INV 17: totalAssetWithdrawn must always be larger than premiumAmount

INV 18: Only owner or delegatee of owner can settle unexpired option

INV 19: approvedPool must only be primePool

INV 20: putAsset must always be a USD denominated asset

INV 21: optionIds must be incrementing each mintOption and split

INV 22: Whenever an option is settled, contract must always hold assetToUse amount which has been received during option minting

Privileged Functions

- transferOwnership
- renounceOwnership
- updatePoolApporvals
- updatePoolSettings
- setApprovedSwapper
- emergencyWithdraw

Issue_01	Governance Issue: Full control over funds
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The owner can withdraw all funds from active options via the emergencyWithdraw function.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_02	Contract is drainable due to incorrect “in-range-check”
Severity	**High**
Description	The previous iteration of the codebase has been exploited by a sophisticated attack which leveraged truncation on the upper/lower boundaries due to high tokenX/tokenY pricing to achieve a seemingly “single-sided” position which in reality was double sided. This then resulted in withdrawing liquidity from a double sided position while the options contract only received tokenX (call) OR tokenY (put) or vice versa during the mintOption function. The received amount however reflects a double sided position which means the missing counter token due to truncation does not flow into consideration. Due to this fact the received amount was smaller than what will be later used as amountToSwap. Whenever an option is now settled, the amountToSwap variable is calculated based on the tick range but always assuming single-sided calculation, which means in this specific scenario, the amountToSwap will always be larger than what has been received during option minting: *uint256 amountToSwap = isAmount0* *? LiquidityAmounts.getAmount0ForLiquidity(* *opTick.tickLower.getSqrtRatioAtTick(),* *opTick.tickUpper.getSqrtRatioAtTick(),* *uint128(liquidityToSettle)* *)* Thus, an inconsistency between what has previously been received by the option contract during mintOption and how much is now intended to be added back to liquidity arises with the deviation towards adding back more funds than what was received. This flaw can be exploited in multiple different variations. To prevent this attack, the mintOption function implements the following validation which *should* ensure that the liquidity is indeed

single-sided and therefore the correct amount is being received by the option contract during liquidity removal:

```
if (
_params.tickLower <
TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))
        && _params.tickUpper >
TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))
) {
revert NotValidStrikeTick();
}
```

This check is however inaccurate, as it is based on ticks instead of sqrtPriceX96. This can result in a position being considered as out-of-range while factually being in range. For example:

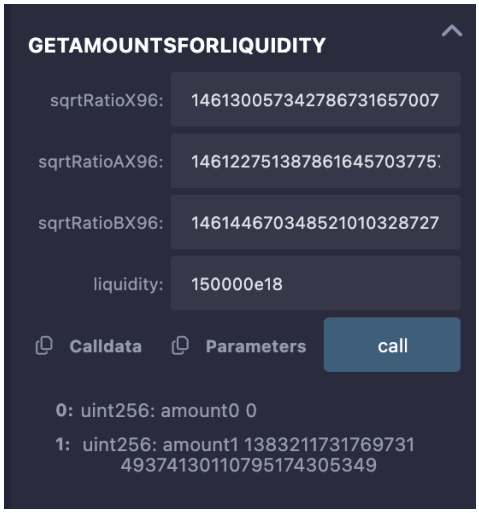
tickLower = currentTick < tickUpper is considered as out-of-range based on the tick check. However, since price -> tick is always rounded down it is possible that lowerPrice < currentPrice which then in turn requires tokenY as well to be in the range. This can be further leveraged with a very high upperTick to fulfill the required properties such as received amount < amountToSwap:

ETH/USDC

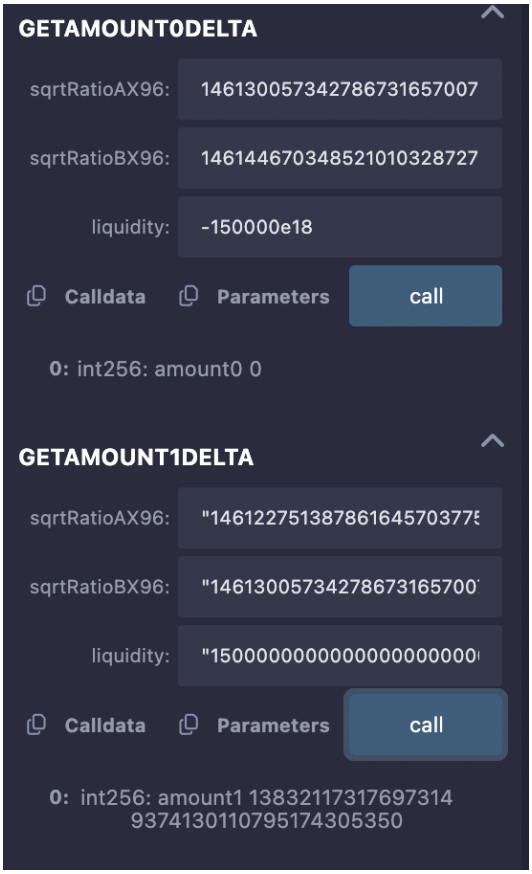
- call = ETH
- put = USDC
- range = [1461227513878616457037757440299078829107947336294; 1461446703485210103287273052203988822378723970341]
- liquidity = 150000e18
- priceBeforeAddAndMint = 1461300573427867316570072651998408279850435624080
- priceBeforeSettle = 1461227513878616457037757440299078829107947336294

a) Swap price to 1461300573427867316570072651998408279850435624080

b) Add liquidity to the range which is currentTick = lowerTick and currentPrice < lowerPrice so it is technically double sided but the validation passes because it erroneously uses tick instead of sqrtX96 price. We now add 1383211731769731493741301107951743053 49 tokenY and tokenX = 0



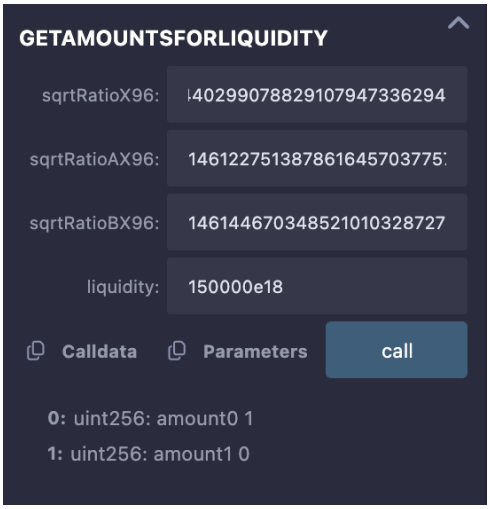
c) Now execute mintOption with the provided liquidity which passes. The reason why it passes is due to the rounding direction when removing liquidity, as getAmount0Delta rounds amountX down when liquidity is removed:



The option contract receives 13832117317697314937413011079517430 5349 tokenY and 0 tokenX (due to above described rounding). This allows for passing the validation within the mintOption function:

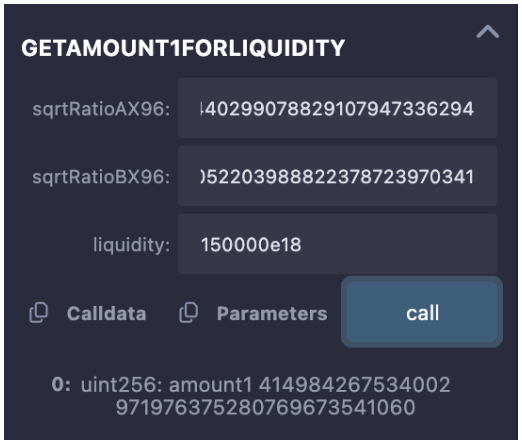
```
} else {
    require(amounts[1] > 0 && amounts[0] == 0);
    amountsPerOptionTicks[i] = (amounts[1]);
    totalAssetWithdrawn += amounts[1];
    isAmount0 = false;
}
```

d) Now swap price down to 1461227513878616457037757440299078829107947336294 (which is the equal to the lower boundary) to get tokenY = 0 and tokenX != 0.



(this is actually 2 due to rounding up when consulting getAmount0Delta)

This furthermore ensures the single-sided property is true which now allows for settlement. It calculates now amountToSwap which is 414984267534002971976375280769673541060 and thus significantly higher than what was initially received during step c), because the single-sided calculation is used.



We now take out this full amount via the swapper (under the premise that it is existing on the options contract) and provide 1 tokenX to the option contract which is exactly amountReq.

Liquidity is added back and the user made a profit from at least (414984267534002971976375280769673541060 -

	13832117317697314937413011079517430534 9) = 27666309435702982260224516997449923571 1. This demonstrates how the validation check within mintOption is bypassed and the exploit is still fully executed. *The exploit can be triggered with many different values and some may succeed or not based on different roundings during liquidity addition or removal. But the root-cause is always the same and the actual PoC is not that important. Summarized, the following properties need to be true for PUT exploit and tokenY = putAsset while bypassing the input validation within mintOption and ensuring the input validation within settleOption passes: > A range and price must be found which allows for double sided liquidity but tokenX = 0. In this range currentTick = lowerTick AND currentPrice < lowerPrice. This bypasses the erroneous check which is based on ticks instead of sqrtX96Price > Option minting must happen at this range and price and delta amount for tokenX must be zero while it is still in-range > The same range must correspond to tokenY = 0 and tokenX != 0 if price is at lowerTick, which then allows for settling. This range must be lowerTick = currentTick > onSwapReceiver contract must allow for P2P/arbitrary execution (example: onSwapReceiver/oneInchSwapper contract) > position must have non-zero tokenX but zero tokenY when currentPrice = lowerPrice > (potentially small unconsumed approvals due to rounding up when adding liquidity within getAmount0/1Delta) The only fee which was paid by the user is the swap fee and all swaps are executed while liquidity remains constant. The first swap was done at the beginning. Then liquidity is added and immediately removed which does not impact swaps. Afterwards a swap down is executed with the same liquidity as before and then liquidity is being added on the right side of the range which allows it to smoothly swap down to the initial tick without any loss of funds.

	Further different iterations of this exploit for example in the following callpath: *if (isAmount0 && ac.isSettle == true) {* *ac.assetToUse.approve(address(positionManager), amount0);* *ac.assetToGet.approve(address(positionManager), amount1);* *uint256 actualAmount0 = LiquidityAmounts.getAmount0ForLiquidity(* *opTick.tickLower.getSqrtRatioAtTick(),* *opTick.tickUpper.getSqrtRatioAtTick(),* *uint128(liquidityToSettle)* *);* *ac.assetToGet.transferFrom(msg.sender, address(this), amount1);* *ac.assetToUse.transfer(msg.sender, actualAmount0 - amount0);* where actualAmount0 > received during mint, are possible.
Recommendations	Consider implementing a check which is based on sqrtPriceX96 instead of ticks. This check must use the current pool price and the price of tickLower/tickUpper. It must not be derived from the current pool tick as this is inaccurate. Furthermore, consider implementing a uniform boundary check for adding liquidity and creating positions which prevents such truncation scenarios in the first place. For example a boundary limitation of upperTick and lowerTick to be at maximum/minimum XXX, which is settable by governance and mimics reasonable market prices for the primePool. This will ensure that a position is indeed single sided.

	Moreover, hooks must be whitelisted to not allow for arbitrary execution and a hysteresis check must be implemented. As a last safeguard, consider implementing a reasonable lower threshold for how much liquidity can be used within mintOption for each sub-option. *It has to be noted that the above recommended validation is intrusive and requires additional time-allocation which goes beyond the included resolution round.*
Comments / Resolution	Resolved: a) A new check has been implemented which is now based on the price of a pool. The following examples are allowed: - X-sided: curPrice = 10; lowerPrice = 10; upperPrice = 15 - Y-sided: curPrice = 10; lowerPrice = 5; upperPrice = 10 It has to be noted that this is a sufficient check to ensure single-sided liquidity. However, an even more safe check would be to remove the ability to meet the exact curPrice with the boundary. The examples from above could look as follows then: - X-sided: curPrice = 10; lowerPrice = 11; upperPrice = 15 - Y-sided: curPrice = 10; lowerPrice = 5; upperPrice = 9 Indeed, it has to be mentioned that due to the remaining existing tick-check: *(opTick.tickLower < TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool)) && opTick.tickUpper > TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool)))*

	, the scenario of Y-sided position remains more restrictive because the price > tick conversion is truncated and thus if the price is not exactly at a tick, it will not be allowed to set upperTick to curTick. b) Boundary for upper/lower tick has been implemented c) Hooks must now be whitelisted d) A lower threshold was implemented However, **the important hysteresis-check is missing**, instead, the mintOption function is made permissioned. Due to the fact that this function was made permissioned, this issue is fully resolved under the assumption that the caller will act in good faith. If the function is made permissionless in the future, we highly recommend implementing a hysteresis-check.

Issue_03	Contract is drainable due to bypassing “in-range-check” with onPositionUse hook
Severity	High
Description	The root-cause of the previously mentioned exploit lies within the fact that a malicious option is being minted which is double-sided at the time of minting. This is partially prevented with the following check: *if (* *_params.tickLower < TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))* *&& _params.tickUpper > TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))* *) {* *revert NotValidStrikeTick();* *}* Even if it would be fully prevented by using a sqrtPrice based check, it would still be exploitable by leveraging the onUsePositionBefore hook. The attack would unfold as follows: a) Add liquidity as usual via the handler with tickUpper being very high/low (depending on CALL/PUT) and the specific crafted range and amount b) Call mintOption with PUT or CALL for the corresponding liquidity. The above mentioned validation passes because at this point the liquidity is indeed single sided. c) During onPositionUseBefore, execute a swap to the left side which ensures that the position has the exploitable properties such as: tokenX = zero due to rounding; position being double-sided and thus tokenY not showing the full required amount. (or vice-versa) d) The liquidity is now removed and the options contract receives less tokenY due to the fact that its single sided and tokenX is rounded to zero (or vice-versa)

	The exploit can now be executed similarly as before.
Recommendations	Consider only allowing for whitelisted hooks.
Comments / Resolution	Resolved, mintOption only allows for using positions with approved hooks.

Issue_04	Strike price calculation is incorrect in case pair is inverse, impacting premium calculation
Severity	**High**
Description	Currently, the strike calculation within the mintOption function is as follows: *uint256 strike = getPricePerCallAssetViaTick(primePool, _params.isCall ? _params.tickUpper : _params.tickLower);* ... *if (sqrtPriceX96 <= type(uint128).max) {* *uint256 priceX192 = uint256(sqrtPriceX96) * sqrtPriceX96;* *price = callAsset == _pool.token0()* *? FullMath.mulDiv(priceX192, 10 ** callAssetDecimals, 1 << 192)* *: FullMath.mulDiv(1 << 192, 10 ** callAssetDecimals, priceX192);* *}* Simplified, this just gives the following conditions for strike prices, in points b) and c) we have included theoretical examples in the scenario where someone wants to bet on the USDC price to increase: **... ticks will be converted to real price for simplicity ...** tickLower = 1500 currentTick = 2500 tickUpper = 3500 **a) ETH/USDC; callAsset = token0** > use tickUpper for strike calculation > strike = tickUpper > strike = 3500 > This is correct, as callAsset is ETH and strike price must be above current price

########## INVERSE PAIR ##########

tickLower = 0.000285
currentTick = 0.0004
tickUpper = 0.00066

b) USDC/ETH; callAsset = token1

> use tickUpper for strike calculation
> strike = tickUpper
> strike = 1/0.00066 = 1515
> This is incorrect, as callAsset is ETH and strike price must be above current price. It is indeed above current price but since the price is inverse, it means it is below current price.

Furthermore, the lastPrice is based on the current SPOT price of the pool. An attacker can thus further increase the price or decrease the price (in the corresponding direction to ensure single-sided property remains) and therefore trigger unexpected side-effects.

The actual root-cause is simple: In case of inverse pair, upperTick does not anymore correspond to the strike price.

***Note that the premium calculation itself is out of scope but the current mechanism strongly indicates that the application during the inverse pair which results in an incorrect strike price will incorrectly influence the premium calculation:**

*volatility = getVolatility(_params.optionsMarket, **_params.strike**, _params.lastPrice, volatility);*

```
        uint256 optionPrice = BlackScholes.calculate(
        _params.isPut ? 1 : 0, _params.lastPrice, _params.strike,
timeToExpiry, 0, volatility
        ) // 0 - Put, 1 - Call
        // Number of days to expiry mul by 100
```

	.div(BlackScholes.DIVISOR);
Recommendations	**In the short term**, consider not supporting inverse pairs where the USD denominated asset is token0. **In the long term**, consider refactoring the logic and undergo a full audit of the premium amount calculation which covers all edge-cases.
Comments / Resolution	Acknowledged. The client indicates that only pairs are used where tokenY is the USD asset. The current protocol logic does not support pairs if tokenX is the USD asset.

Issue_05	In-range validation within mintOption is based on _params.tickLower/Upper instead of opTick.tickLower/Upper
Severity	High
Description	Currently, the mintOption function executes the following in-range-check to ensure a position is not double sided: *if (* *_params.tickLower < TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))* *&& _params.tickUpper > TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))* *) {* *revert NotValidStrikeTick();* *}* Apart from it being inaccurate due to the usage of tick instead of sqrtPriceX96, it is based on _params and not on opTick. Therefore a user can simply bypass this check completely since it only relies on _params and opTick.tickLower/Upper remains completely unvalidated, as only the strike tick validation is executed: *if (_params.isCall ? _params.tickUpper != opTick.tickUpper : _params.tickLower != opTick.tickLower) {* *revert NotValidStrikeTick();* *}* A user can for example provide: opTick.tickLower < tickCurrent < opTick.tickUpper while at the same tick: _params.tickLower > currentTick < _params.tickUpper

	The root-cause of this issue is the lack of validation between opTick.tickLower/tickUpper and _params.tickLower/tickUpper
Recommendations	Consider ensuring that sub-options have the same tickLower and tickUpper value as the main option. If that is true, there is no issue with using _params.tickLower/tickUpper for validation purposes.
Comments / Resolution	Resolved, it is now ensured that sub-options have the exact same tick range as the main option.

Issue_06	mintOption and settleOption can be called with manipulated price
Severity	**High**
Description	Both functions do not currently execute a hysteresis check. This allows malicious users to invoke these functions while the pair is in a manipulated state. Multiple potential issues can arise from this flexibility and this is furthermore an additional required property for at least two identified issues: a) Contract is drainable due to incorrect “in-range-check” b) Frontrun of settlement with a swap allows for tricking settler into unexpected call-path c) Manipulation of premium calculation as currentTick flows into the volatility determination. The same counts for minOptionPrice calculation based on the current spot price. When a call option is minted a user can swap the spot price to the right side which then results in a very small minOptionPrice (OOS property). Additionally, if such a check is invoked at the beginning of the function, it is then possible to still shift the price of a pool during hooks. Therefore we recommend only allowing for whitelisted hooks which prevent arbitrary execution. This will also fix other multiple issues which are related to hooks.
Recommendations	Consider implementing a hysteresis check preferably based on a trusted oracle which reverts if the spot price of the pair deviates a certain percentage from the real price. Additionally it is required to only allow for creating positions with whitelisted hooks. *The validation of this fix requires additional time-allocation which goes beyond the included resolution round.*
Comments / Resolution	Acknowledged. The client has changed the mintOption function to being permissioned. In case this change is reverted, we highly recommend implementing a hysteresis-check.

Issue_07	Incorrect premiumAmount calculation in specific scenario
Severity	High
Description	The getOptionPrice function is out of scope for this audit, therefore we use minOptionPrice as 90% of the lastPrice for our examples. The codebase exposes 4 possible paths when a strike price calculation can happen, these will be as follows: Core assumption: getOptionPrice returns price in current pair denomination - ETH = 18 decimals - USDC = 6 decimals - ETH/USDC = 2500e-12 - USDC/ETH = 0.0004e12 - option prices are 90% of the real prices (for simplicity) ############## CALL execution ################# SCENARIO 1: ETH/USDC; token0 = CALL - price = 2500e-12 - getOptionPrice = 2250e-12 - premiumInQuote = withdrawAmount * getOptionPrice / callAssetDecimals - amount = premiumInQuote * callAssetDecimals / lastPrice - premiumInQuote = 1e18 * 2250e-12 / 1e18 - premiumInQuote = 2250e-12 (2250 USDC) - amount = 2250e-12 * 1e18 / 2500e-12 - amount = 0.9e18 (0.9 ETH) SCENARIO 2: USDC/ETH; token1 = CALL - price = 2500e-12 - getOptionPrice = 2250e-12 - premiumInQuote = withdrawAmount * getOptionPrice / callAssetDecimals - amount = premiumInQuote * callAssetDecimals / lastPrice

- premiumInQuote = 1e18 * 2250e-12 / 1e18
- premiumInQuote = 2250e-12
- amount = 2250e-12 * 1e18 / 2500e-12
- amount = 0.9e18 (0.9 ETH)

As we can see, for both CALL execution paths, the premiumAmount is 90% of the withdrawn amount, which correctly reflects the minOptionPrice assumption of 90% from lastPrice.

Now whenever a PUT execution is triggered, amount will be divided by strike:

*_params.isCall ? totalAssetWithdrawn : (totalAssetWithdrawn * (10 ** putAssetDecimals)) / strike*

Applying this logic to the formal verification, we will receive an incorrect output for premiumAmount:

############ PUT execution #################

SCENARIO 3: ETH/USDC; token1 = PUT

- price = 2500e-12
- getOptionPrice = 2250e-12
- withdrawAmount = totalAssetsWithdrawn * putAssetDecimals / strike
- premiumInQuote = withdrawAmount * getOptionPrice / putAssetDecimals
- withdrawAmount = 1250e6 * 1e6 / 1500e-12
- withdrawAmount = 8.3e23
- premiumInQuote = 8.3e23 * 2250e-12 / 1e6
- premiumInQuote = 1867.5e6

As illustrated- premiumInQuote is larger than the initial withdrawn amount. This is incorrect.

Contrary to that, if the lastPrice would be used as divisor:

	- price = 2500e-12 - getOptionPrice = 2250e-12 - withdrawAmount = totalAssetsWithdrawn * putAssetDecimals / strike - premiumInQuote = withdrawAmount * getOptionPrice / putAssetDecimals - withdrawAmount = 1250e6 * 1e6 / 2500e-12 - withdrawAmount = 5e23 - premiumInQuote = 5e23 * 2250e-12 / 1e6 - premiumInQuote = 1125e6 The correct premiumInQuote value is reflected, which is 90% of the withdrawn amount. This example shows that the usage of strike as divisor is incorrect OR the usage of minOptionPrice in that scenario is incorrect.
Recommendations	Consider refactoring the premium calculation. Afterwards a full audit of the whole premium calculation logic should be conducted, including the current OOS logic.
Comments / Resolution	Not resolved. The client disagreed with this issue as follows: ########## *"Just to clarify — for put options, we should not divide the withdrawn assets by the spot (last) price, as it leads to incorrect behavior.* *Consider this example:* *A user wants to purchase a put option with the following parameters:* *• Strike price: 2500* *• Size exposure: 10,000 USDC* *• Spot price: 3000* *• This corresponds to 4 options, since:*

	10,000 ÷ 2500 (strike) = 4 contracts *The correct logic is implemented as:* *_params.isCall* *? totalAssetWithdrawn* *: (totalAssetWithdrawn * (10 ** putAssetDecimals)) / strike* *However, if we divide by the spot price instead:* *10,000 ÷ 3000 = 3.33 options* *This underestimates the number of contracts.* *Why this is incorrect:* *In options, the strike price determines the notional exposure — not the current spot price.* *For example:* *If a user wants to buy 100 options at a strike of 2000, they are seeking $200,000 in exposure:* *100 options × 2000 strike = $200,000* *This calculation should remain the same, regardless of whether the current price is 1500, 2500, or 3000."* *##########* To respond to this, we first need to clarify that the OptionPricing logic is fully out of scope. Therefore, to provide a proper PoC, we consider the following function: *function _getOptionPrice(OptionPriceParams memory _params) internal view returns (uint256) {* *uint256 timeToExpiry = _params.expiry.sub(block.timestamp).div(864);*

```
        uint256 volatility = ttlToVol[_params.optionsMarket][_params.ttl];

        if (volatility == 0) revert Vol_Not_Set();

        volatility = getVolatility(_params.optionsMarket, _params.strike,
_params.lastPrice, volatility);

        uint256 optionPrice = BlackScholes.calculate(
            _params.isPut ? 1 : 0, _params.lastPrice, _params.strike,
timeToExpiry, 0, volatility
        ) // 0 - Put, 1 - Call
              // Number of days to expiry mul by 100
            .div(BlackScholes.DIVISOR);

        uint256 minOptionPrice =
_params.lastPrice.mul(minOptionPricePercentage[_params.optionsM
arket]).div(1e10);

        if (minOptionPrice > optionPrice) {
            return minOptionPrice;
        }

        return optionPrice;
    }
```

More specifically, the following snippet of this function is important:

```
        uint256 minOptionPrice =
_params.lastPrice.mul(minOptionPricePercentage[_params.optionsM
arket]).div(1e10);

        if (minOptionPrice > optionPrice) {
            return minOptionPrice;
        }
```

This means, we consider the minOptionPrice as a valid return value in *all* scenarios. This price is based on the current price of the pair

	and is considered as a source of truth for our scenario and to demonstrate the issue which we have explained. Now we reiterate the issue with a new/additional/simplified example which showcases the incorrect premiumAmount calculation: ########################## ETH/USDC; token1 = PUT ETH price = 2500e6 USDC price = 1e6 pairPrice = 2500e6 strikePrice = 2000e6 (PUT option = strikePrice below currentPrice) totalAssetsWithdrawn = 1000e6 amount (based on isCall = false) = 0.5e6 *1e6 is only for illustration purposes. This scaling can be completely ignored as it will be the same result with all other scales. The above setup is a valid setup, we will now demonstrate based on this setup the premiumAmount calculation based on the minOptionPrice (90% of currentPrice) as source of truth: - getOptionPrice returns minOptionPrice (as explained above). Thus, getOptionPrice = 2250e18 - premiumInQuote = amount * minOptionPrice / 1e6 - premiumAmount = 0.5e6 * 2250e6 / 1e6 - premiumAmount = 1125e6 - Due to short circuit during put asset calculation, premiumAmount = 1125e6 As one can see, it is not practicable to apply a premiumAmount of 1125e6 if the withdrawnAmount is 1000e6. As previously mentioned, the OptionPricing logic is fully out of scope, hence, the comment of the developer may remain true and the flaw may thus be from an incorrect minOptionPrice value within

	the out of scope contract, while this issue may not apply if the short circuit path within _getOptionPrice is not triggered: *if (minOptionPrice > optionPrice) {* *return minOptionPrice;* *}* However, as we have delivered proof that in this scenario, the premiumAmount is wrong, we still consider this issue as being valid. The issue headline has been adjusted to reflect the developers input.

Issue_08	Frontrun of settlement with a swap allows for tricking settler into unexpected call-path
Severity	High
Description	Within settleOption, there are multiple different call-paths which allow for different executions based on the current state of the option (OTM; partially ITM, ITM). All executions are trivially manipulatable by an attacker to steal a partial share or full share of the profit. **Scenario 1:** If for example a user wants to settle an expired option which is fully OTM, a malicious user can frontrun this settlement with a swap which brings the option into partial profit based on the current price of the pair. More specifically, the following callpath would be triggered: *else {* *if (isAmount0 && ac.isSettle == true) {* *ac.assetToUse.approve(address(positionManager), amount0);* *ac.assetToGet.approve(address(positionManager), amount1);* *uint256 actualAmount0 = LiquidityAmounts.getAmount0ForLiquidity(* *opTick.tickLower.getSqrtRatioAtTick(),* *opTick.tickUpper.getSqrtRatioAtTick(),* *uint128(liquidityToSettle)* *);* *ac.assetToGet.transferFrom(msg.sender, address(this), amount1);* *ac.assetToUse.transfer(msg.sender, actualAmount0 - amount0);* *}*

	This would result in the user transferring in tokenY while at the same time assuming tokenX is worth more due to the fact that the pair shows an incorrect price. The settler would essentially be tricked into supplying tokenY at the wrong price, resulting in a loss for the settler and profit for sandwich-attacker. **Scenario 2:** Similar to this it is also possible to frontrun ITM settling by swapping towards the pair state being double sided, now the settler is required to partially add liquidity via a transfer-from and the sandwich attacker can gain a profit by swapping back. **Scenario 3:** A sandwich attacker can trivially swap the price down in case the option is ITM to then make the option fully OTM. This results in liquidity being added back while the pair is at the wrong tick, allowing the sandwich attacker to steal the full option profit by swapping back.
Recommendations	Consider implementing a hysteresis check as well as allowing to provide a slippage parameter to ensure the current price does not deviate too much from the expected price before the transaction execution. *The validation of a hysteresis-check requires additional time-allocation which goes beyond the included resolution round.*
Comments / Resolution	Acknowledged, the client indicated that settleOption will be fully permissioned and that these will be executed with a guard contract that implements corresponding safety measurements.

Issue_09	Sub-options can have different strike price in case of inverse pair
Severity	**Medium**
Description	In the issue : “Strike price calculation is incorrect in case pair is inverse, impacting premium calculation”, we have already elaborated that the following strike price calculation for inverse pairs is incorrect: *uint256 strike = getPricePerCallAssetViaTick(primePool, _params.isCall ? _params.tickUpper : _params.tickLower);* The same blunder is existing in the snippet which validates the uniform strike price among all sub-options: *if (_params.isCall ? _params.tickUpper != opTick.tickUpper : _params.tickLower != opTick.tickLower) {* *revert NotValidStrikeTick();* *}* Therefore, for an inverse pair (USDC/ETH), it is possible to create sub-options with different strike prices, as only tickUpper is validated which is however not the strike price (tickLower is): Option Lifecycle USDC/ETH  This is a clear invariant violation and will result in side-effects during settleOption as now some sub-options can be ITM while other sub-options can be OTM. Unexpected side-effects are to expected. Bailsec will not further investigate the impact of the issue as it must be clearly fixed. (**No inverse pairs should be used** or the logic must be fully refactored)
Recommendations	In the short term, consider not supporting inverse pairs where the USD denominated asset is token0.

	In the long term, consider refactoring the logic and undergo a full audit of the premium amount calculation which covers all edge-cases. Moreover, the lowerTick and upperTick both should be validated to ensure sub-options always match values from the optionId.
Comments / Resolution	Resolved. The client indicated that no inverse pairs are not supported. This means any pair with token0 = USD denominated asset and token1 = WETH/WSONIC/... will not be supported. Moreover, it is now validated that sub-options have the same lower/upper tick as the main option: *if ((_params.tickUpper != opTick.tickUpper || _params.tickLower != opTick.tickLower)) {* *revert NotValidStrikeTick();* *}*

Issue_10	settleOption does often not add full liquidity back
Severity	Medium
Description	The settleOption function re-adds liquidity to the handler. However, as the liquidity process re-calculates the liquidity which is being added based on the provided amounts: *{* *(uint160 sqrtPriceX96,,,,,,) = pool.slot0();* *uint160 sqrtRatioAX96 = TickMath.getSqrtRatioAtTick(params.tickLower);* *uint160 sqrtRatioBX96 = TickMath.getSqrtRatioAtTick(params.tickUpper);* *liquidity = LiquidityAmounts.getLiquidityForAmounts(* *sqrtPriceX96, sqrtRatioAX96, sqrtRatioBX96, params.amount0Desired, params.amount1Desired* *);* *}* It will almost always result in less liquidity being added than expected, the dust will remain in the PositionManager contract. This can be abused by a malicious actor via first minting an option and then settling it in multiple small steps such that each liquidity addition will result in a loss. This is specifically problematic since the settling could be made permissionless in the future. This will also result in an erroneous decrease of liquidityToUse: *positionManager.unusePosition(opTick._handler, unusePositionData);* *opTick.liquidityToUse -= liquidityToSettle;* Because the return value of unusePosition may be smaller than liquidityToSettle. However, we remain unsure if the return value should be casted and used to decrease liquidityToUse, as this could

	potentially introduce other side effects (even though being technically correct).
Recommendations	Consider ensuring settling remains whitelisted only. Since this will however only prevent the expiry state exploit and not the ITM state, it is recommended to implement an additional variable during option minting which determines a minimum settling threshold based on the used liquidity to avoid that a malicious actor partially settles 1000 times whereas each time liquidity is being lost. This could be for example 10% of the used liquidity during minting. *The validation of this fix requires additional time-allocation which goes beyond the included resolution round.*
Comments / Resolution	Acknowledged.

Issue_11	Insufficient in-range validation within settleOption
Severity	**Medium**
Description	Similar to the very first issue, settleOption exposes insufficient in-range validation which allows for edge-case positions. In the scenario where such a position is being settled, the amount0 or amount1 return value which is non-zero (indeed, one of both will be zero) will be decreased compared to as-if the single sided calculation would have been executed (getAmount0ForLiquidity/getAmount1ForLiquidity). This can result in erroneous callpaths to be triggered which should not be triggered.
Recommendations	Consider implementing the correct validation which relies on sqrtPriceX96 as well as implementing a hysteresis check. Furthermore, consider only allowing for whitelisted hooks which prevent arbitrary execution such as price changes before liquidity addition. *The validation of a hysteresis-check requires additional time-allocation which goes beyond the included resolution round.*
Comments / Resolution	Resolved, a validation based on the price instead of the tick has been implemented and furthermore this function is now fully permissioned. In case the function is made permissionless, we highly recommend implementing a hysteresis check.

Issue_12	settleOption lacks explicit revert in case of untriggered callpath
Severity	**Medium**
Description	The settleOption function has multiple different callpaths which are depending on the option range and the current price. Currently, there is no explicit revert in case no callpath is met. This is for example the case if the option is: > not expired > not ITM > option owner/delegatee is the caller In that scenario, no callpath will be executed but the Handler.unusePosition function is still being triggered. In many scenarios it would just revert because approvals are missing. However, as we have already explained it is possible that approvals are not always consumed due to the truncation in the premiumAmountEarned calculation. This would then indeed allow the Handler.unusePosition function to succeed and re-add liquidity. This is a highly invalid state and can result in many unexpected edge-cases. This could even be exploited in combination with a sandwich attack to add tokens around the wrong tick for unconsumed approvals.
Recommendations	Consider explicitly reverting if no callpath is triggered.
Comments / Resolution	Resolved. While this issue has been technically not resolved, the settleOptions function is made permissioned, which mitigates this possible attack-path. In the case where this function is made permissionless, we recommend solving this issue. It has to be noted that any permissioned address can still exploit this.

Issue_13	Usage of primePool for swapper can push position into invalid state
Severity	Medium
Description	Whenever an option is ITM and settled, the used asset is fully being swapped to the required asset. This will have an inherent price impact on the primePool. In many cases it will result in pushing the current price within the range which then means tokenX and tokenY are required which then reverts upon liquidity addition. Furthermore it introduces a pathway to force ratios to be different which can then be exploited in case there are outstanding approvals. In an edge-case this could result in amountReq to be above than what is indeed added back if the position is pushed into the known invalid state after a swap where it is in-range but only consists of one token. Therefore the liquidity addition would consume less than expected which results in a significant amount of unconsumed approvals. These unconsumed approvals can then be tricked out of the contract as explained in another issue.
Recommendations	Consider thinking about further risks from this implementation. **We highly recommend implementing the same (fixed) in-range check which is existing at the beginning to the line after the swap. This will ensure that no unexpected side-effects can occur from a swap and that the position will always stay in a valid state post-swap.**
Comments / Resolution	Resolved. While this issue has been technically not resolved, the settleOptions function is made permissioned, which mitigates this possible attack-path. In the case where this function is made permissionless, we recommend solving this issue. It has to be noted that any permissioned address can still exploit this.

Issue_14	Invalid callpath in case option is not expired but OTM
Severity	Medium
Description	Whenever an option is OTM, it means that the hypothetical liquidity requires the exact same amount of tokenX or tokenY which was withdrawn during option minting. This is usually handled in the following callpath: *if (isAmount0 && amount0 > 0 && ac.isSettle == true) {* *ac.assetToUse.approve(address(positionManager), amount0);* *ac.totalAssetRelocked += amount0;* *}* However, this callpath is only being triggered in the scenario where an option is expired because it requires ac.isSettle == true. If now the settleOption function is called while the option is OTM and at the same time it is not expired, the following callpath is erroneously triggered: *else {* *uint256 amountToSwap = isAmount0* *? LiquidityAmounts.getAmount0ForLiquidity(* *opTick.tickLower.getSqrtRatioAtTick(),* *opTick.tickUpper.getSqrtRatioAtTick(),* *uint128(liquidityToSettle)* *)* *: LiquidityAmounts.getAmount1ForLiquidity(* *opTick.tickLower.getSqrtRatioAtTick(),* *opTick.tickUpper.getSqrtRatioAtTick(),* *uint128(liquidityToSettle)* *);* *ac.totalAssetRelocked += amountToSwap;* *uint256 prevBalance = ac.assetToGet.balanceOf(address(this));*

```
                if (!approvedSwapper[address(_params.swapper[i])]) {
                revert NotApprovedSwapper();
                }

                ac.assetToUse.transfer(address(_params.swapper[i]),
amountToSwap);

                _params.swapper[i].onSwapReceived(
                address(ac.assetToUse), address(ac.assetToGet),
amountToSwap, _params.swapData[i]
                );

                uint256 amountReq = isAmount0
                ? LiquidityAmounts.getAmount1ForLiquidity(
                opTick.tickLower.getSqrtRatioAtTick(),
                opTick.tickUpper.getSqrtRatioAtTick(),
                uint128(liquidityToSettle)
                )
                : LiquidityAmounts.getAmount0ForLiquidity(
                opTick.tickLower.getSqrtRatioAtTick(),
                opTick.tickUpper.getSqrtRatioAtTick(),
                uint128(liquidityToSettle)
                );

                uint256 currentBalance =
ac.assetToGet.balanceOf(address(this));

                if (currentBalance < prevBalance + amountReq) {
                revert NotEnoughAfterSwap();
                }

                ac.assetToGet.approve(address(positionManager),
amountReq);

                ac.totalProfit += currentBalance - (prevBalance +
amountReq);
        }
```

	This callpath is however only meant to be triggered if an option is ITM. Usually, it would just revert because the swap will never be sufficient to honor the amountReq and even if it would be honored, the liquidity addition would revert most of the time if no unconsumed approvals for the useAsset is existing. However, there is still the possibility for unexpected side-effects.
Recommendations	Consider implementing an explicit revert for this scenario.
Comments / Resolution	Resolved. While this issue has been technically not resolved, the settleOptions function is made permissioned, which mitigates this possible attack-path. In the case where this function is made permissionless, we recommend solving this issue. It has to be noted that any permissioned address can still exploit this.

Issue_15	Unsafe downcasting of liquidity can be used to exploit storage in case different handler is being used
Severity	Medium
Description	Within UniswapV3, liquidity must not be larger as uint128. This is ensured throughout the codebase as in multiple spots the liquidity is downcasted to uint128 whenever interactions with Uniswap libraries are happening. This can be seen here for example: *ac.assetToUse.approve(address(positionManager), amount0);* *ac.assetToGet.approve(address(positionManager), amount1);* *uint256 actualAmount0 = LiquidityAmounts.getAmount0ForLiquidity(* *opTick.tickLower.getSqrtRatioAtTick(),* *opTick.tickUpper.getSqrtRatioAtTick(),* *uint128(liquidityToSettle)* *);* *ac.assetToGet.transferFrom(msg.sender, address(this), amount1);* The problem here is that users are always allowed to provide a liquidity parameter up to uint256, as can be seen in the parameter structs: *struct SettleOptionParams {* *uint256 optionId;* *ISwapper[] swapper;* *bytes[] swapData;* *uint256[] liquidityToSettle;* *}* *struct PositionSplitterParams {* *uint256 optionId;* *address to;*

```
uint256[] liquidityToSplit;
}


struct OptionTicks {
IHandler _handler;
IUniswapV3Pool pool;
address hook;
int24 tickLower;
int24 tickUpper;
uint256 liquidityToUse;
}
```

This opens several different attack-paths and unexpected side-effects.

A user can for example provide OptionParams.optionTicks.liquidityToUse which is larger than uint128. It will then save this liquidity amount for the sub-option:

```
opTickMap[optionIds].push(
OptionTicks{{
    _handler: opTick._handler,
    pool: opTick.pool,
    hook: opTick.hook,
    tickLower: opTick.tickLower,
    tickUpper: opTick.tickUpper,
    liquidityToUse: opTick.liquidityToUse
}}
);
```

With the current handler setup, it reverts during decoding the parameters as soon as liquidity is larger than uint128:

```
(UsePositionParams memory _params, bytes memory hookData) =
    abi.decode(_usePositionData, (UsePositionParams, bytes));
```

However, if a new handler is being implemented without decoding

	safeguards, this issue can be expanded.
Recommendations	Consider ensuring the liquidity in all parameters is a uint128.
Comments / Resolution	Acknowledged.

Issue_16	Bypass option of tick range limitation in case of different signs
Severity	**Medium**
Description	Within the mintOption, the following snippet was introduced: *if (opTick.tickUpper > 0 && opTick.tickLower > 0 ||* *opTick.tickUpper < 0 && opTick.tickLower < 0) {* *if (uint24(opTick.tickUpper - opTick.tickLower) >* *maxTickDiff) {* *revert NotValidStrikeTick();* *}* *} else {* *if (uint24(opTick.tickUpper + opTick.tickLower) >* *maxTickDiff) {* *revert NotValidStrikeTick();* *}* *}* This snippet was implemented to ensure that only positions with reasonable maximum tick range can be minted. Example: tickUpper = 100 tickLower = 0 maxTickDiff = 50 -> revert This snippet is however flawed as it only works properly if both ticks have the same sign. If one of both ticks is < 0, the else clause will be entered: *else {* *if (uint24(opTick.tickUpper + opTick.tickLower) >* *maxTickDiff) {* *revert NotValidStrikeTick();* *}*

	Consider the example: tickLower = -50 tickUpper = 100 the tickRange here is 150. However, the check is based on: 100 + (-50) = 50 This allows an attacker to fully bypass this. Likewise, the following scenario: tickUpper = 5 tickLower = - 10 5 + (-10) = -5 > revert
Recommendations	Consider always comparing the range based on the absolute difference.
Comments / Resolution	

Issue_17	Unconsumed approvals can be added around wrong price by leveraging onUnusePosition hook and manipulating price
Severity	**Low**
Description	Currently, it is possible to trick the contract into requiring different token ratios upon settleOption. This is trivially possible by settling an expired, single sided position (in tokenX for example), while swapping to the right side during the hook which then requires tokenX and tokenY to be provided during the addLiquidity call. Notably, the tokenY amount is provided at the wrong tick which allows then swapping back for a profit. This issue is only rated as low severity because unconsumed approvals are often just dust.
Recommendations	Consider implementing whitelisted hooks.
Comments / Resolution	Resolved due to the implementation of whitelisted hooks.

Issue_18	Minting and settling options does not require minimum liquidity threshold
Severity	**Low**
Description	Currently it is possible to mint and settle options with an arbitrarily low liquidity amount. This allows for potential truncation/rounding scenarios. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider implementing a variable which is settable by governance and ensures a minimum threshold is enforced.
Comments / Resolution	Resolved.

Issue_19	Sub-options ranges can be different from main option
Severity	Low
Description	The mintOption function expects the following parameters as calldata: *struct OptionParams {* *OptionTicks[] optionTicks;* *uint256 ttl;* *uint256 maxCostAllowance;* *int24 tickLower;* *int24 tickUpper;* *bool isCall;* *}* Notably, the optionTicks array is as follows: *struct OptionTicks {* *IHandler _handler;* *IUniswapV3Pool pool;* *address hook;* *int24 tickLower;* *int24 tickUpper;* *uint256 liquidityToUse;* *}* The OptionParams represent the main option which consists of options between tickLower and tickUpper. The OptionTicks however represent sub-options which can be aggregated under a main option. Depending on the direction (PUT/CALL), tickUpper or tickLower is validated for main and sub-option: *if (_params.isCall ? _params.tickUpper != opTick.tickUpper : _params.tickLower != opTick.tickLower) {* *revert NotValidStrikeTick();* *}* However, the other boundary of the range is never validated. This

	means in the scenario of a call option, tickUpper from the sub and main option matches, while tickLower is not validated at all. This means a user can create sub-options which differ in range from the main option, as tickLower can be anything. This is not desired and can result in unexpected side-effects.
Recommendations	Consider validating sub-option ranges against main-option ranges within each loop execution.
Comments / Resolution	Resolved.

Issue_20	Truncation within premiumAmountEarned can result in stuck assetToUse and bypass of fee
Severity	**Low**
Description	The premiumAmountEarned value is calculated as follows: *uint256 premiumAmountEarned = (amountsPerOptionTicks[i] * premiumAmount) / totalAssetWithdrawn;* In the scenario where the totalAssetWithdrawn is larger than (amountsPerOptionTicks[i] * premiumAmount), the result will be zero. While it is true that the caller is required to transfer the full amount in, it is later possible to add the idle token to the wrong tick which then allows for retrieving a part of the lost value. This will result in a net-positive for the user.
Recommendations	Consider reverting in case premiumAmountEarned is zero. Eventual side-effects where this might be a legitimate state should be considered. The option price calculation is happening out of scope within the IOptionPricingV2 contract.
Comments / Resolution	Acknowledged.

Issue_21	Usage of swapper introduces additional user flexibility
Severity	**Low**
Description	Currently it is possible for governance to whitelist different swapper addresses. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic. Therefore, we recommend only adding the primePool for swapping purposes.
Recommendations	Consider only allowing swap via the primePool. While this has the downside that some settlements revert due to price change, it ensures no arbitrary execution can be sneaked in and no manipulated swaps can be executed (for example if another pool is used which has a manipulated price).
Comments / Resolution	Acknowledged. The team indicated that only the primePool will be used as a swapper.

Issue_22	mintOption can be called with multiple duplicate sub-options
Severity	**Low**
Description	The mintOption function allows for the minting of different sub-options. The main reason for this is aggregating liquidity from different DEXes. The current logic allows users to create more than one sub-option for the same DEX in the same range. After our analysis, there is no legitimate reason why one would be doing that and at the same time this can have side-effects during the settleOption function whenever a swap is executed as this swap execution will now impact the subsequent sub-option in the loop if it uses the same pool. This can potentially result in uncovered vulnerabilities.
Recommendations	Consider only allowing one sub-option for tickLower; tickUpper; pool. This can be trivially done using a keccak256 check and revert if such a sub-option for tickLower; tickUpper; pool already exists.
Comments / Resolution	Acknowledged.

Issue_23	Griefing: mintOption can be griefed in case full liquidity usage is desired
Severity	**Low**
Description	The mintOption function allows to create one or multiple suboptions using liquidity from a specific range and pool/hook setting. More specifically it is tied to: *keccak256(abi.encode(addressHandler, _params.pool, _params.hook, _params.tickLower, _params.tickUpper))* If a user wants to mint an option with the full remaining available liquidity for the setting, another user can simply call mintOption with liquidityToUse = 1 wei, which then results in the main call to revert.
Recommendations	Consider acknowledging this issue and include input validation which ensures that liquidityToUse is a specific minimum value. This will greatly decrease user flexibility without compromising UX.
Comments / Resolution	Acknowledged.

Issue_24	Lack of liquidityToSplit check within splitPositions
Severity	**Low**
Description	Currently, there is no check which ensures that an element within the liquidityToSplit[] array is zero. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider explicitly reverting if an element within the liquidityToSplit array is zero. (Do not continue the loop).
Comments / Resolution	Acknowledged.

Issue_25	Off-by-one error in positionSplitter expiration check
Severity	**Low**
Description	Currently, a position is considered as expired as soon as block.timestamp >= expiry: *if (block.timestamp >= oData.expiry) {* *ac.isSettle = true;* *if (!settlers[msg.sender]) {* *revert NotApprovedSettler();* *}* This property is incorrectly checked within the positionSplitter function: *if (oData.expiry < block.timestamp) {* *revert Expired();* *}*

	As one can see, a position is here only considered as expired when block.timestamp > expiry.
Recommendations	Consider following a uniform approach and changing the check to <= .
Comments / Resolution	Resolved.

Issue_26	Delegator cannot split an option
Severity	**Low**
Description	The following error in the positionSplitter function indicates that a delegator should be allowed to split a position: *if (ownerOf(_params.optionId) != msg.sender) {* *revert NotOwnerOrDelegator();* *}* However, this is not the case. It will simply revert in case a legitimate delegator is the caller.
Recommendations	Consider either adjusting the error or also allowing the delegator to split an option.
Comments / Resolution	Acknowledged.

Issue_27	Contract does not support USDT on mainnet
Severity	**Low**
Description	Currently, approvals are made using the normal approve function. For USDT on mainnet, this will revert if an outstanding approval is existing: *function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {* *// To change the approve amount you first have to reduce the addresses`* *// allowance to zero by calling `approve(_spender, 0)` if it is not* *// already 0 to mitigate the race condition described here:* *//* *https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729* *require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));* *allowed[msg.sender][_spender] = _value;* *Approval(msg.sender, _spender, _value);* *}* An attacker can therefore call the mintOption function such that premiumAmountEarned for USDT rounds down which does then not consume the approval. This would essentially break the contract. Furthermore, the emergencyWithdraw function uses transfer instead of safeTransfer.
Recommendations	Consider adhering to the SafeERC20 pattern.
Comments / Resolution	Partially Resolved. Transfers on L501 and L569 are still not using SafeERC20

Issue_28	Full swap execution during ITM settlement is sub-optimal
Severity	**Low**
Description	Currently, whenever an option is settled ITM, the assetToUse is fully swapped to assetToGet and then the leftover amount of assetToGet is transferred to the settler which determines the profit. It has already been clarified that such a swap on the primePool can bring the price within the range which then would revert on liquidity addition. The idea that a full swap must be executed is erroneous, as even with a partial swap of assetToUse the amountReq of assetToGet will be reached. This would have less price impact and would result in less reverts. There would furthermore be no side-effect for the user as it does not make any difference if assetToUse or assetToGet is paid out as profit.
Recommendations	We recommend acknowledging this issue and not manipulating the current swap procedure as this will introduce unexpected side-effects.
Comments / Resolution	Acknowledged.

Issue_29	Incorrect error usage for in-range-check
Severity	**Informational**
Description	Within the mintOption function, the following check and corresponding error is applied: *if (* *_params.tickLower < TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))* *&& _params.tickUpper > TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))* *) {* *revert NotValidStrikeTick();* *}* The error NotValidStrikeTick does not apply to this validation.
Recommendations	Consider adjusting the revert reason.
Comments / Resolution	Acknowledged.

Issue_30	Typographical error(s)
Severity	**Informational**
Description	The contract contains one or more typographical errors: - updatePoolApporvals
Recommendations	Consider fixing the function name.
Comments / Resolution	Acknowledged.

Handlers

V3BaseHandlerVe33Shadow

The V3BaseHandlerVe33Shadow contract is the handler contract for interactions with the Shadow DEX. It essentially is the owner of all active liquidity and allows for various different operations. A whitelist mechanism has been implemented which ensures that only the valid PositionManager can interact with it.

There are two distinct use-cases:

a) **Direct interactions from users via the PositionManager:** This allows users to add and remove liquidity in exchange for a determined ERC6909 token by directly calling PositionManager.mintPosition and PositionManager.burnPosition which then forwards the call to the Handler contract. This also includes the wildcard functionality which will be thoroughly explained in the corresponding appendix.

b) **Indirect interactions from users via the OptionsMarketOTMFE contract:** This allows users to mint and settle options by directly calling OptionMarketOTMFE.mintOption and OptionMarketOTMFE.settleOption. It will then forward the call to the PositionManager and then to the Handler contract which essentially removes liquidity if an option is being minted and re-adds liquidity if an option is being settled. This also applies for the premium donation.

Positions are represented as an ERC6909 token and derived as follows:

> uint256(keccak256(abi.encode(address(this), _params.pool, _params.hook, _params.tickLower, _params.tickUpper)))

The contract furthermore handles the fee accumulation and collection for all positions with active liquidity by re-calculating it on every important state transition and collecting it to the feeReceiver address.

Appendix: Liquidity Split

The _splitPosition function allows users to split positions into multiple sub-positions. It first burns the user's old position for the full range, collecting the corresponding amounts of token0 and token1. Then it checks whether the current price is in range; if so, it reserves a portion specifically for a tiny sub-range around the current price, and distributes the

remaining tokens linearly across the other sub-ranges. After computing how many tokens belong to each new mini-range, it mints new liquidity positions for each of those ranges (using mintInternal), subtracting out the tokens used. Any leftover tokens, which are not needed to fund the new sub-ranges, are returned to the user.

Appendix: Reserve Mechanism

The contract allows for a reserving mechanism in case users want to reserve their current liquidity to be withdrawn as soon as it becomes available/settled.
Under normal circumstances it is possible that a settlement re-adds liquidity which is quickly followed by a new option minting that removes liquidity again.
In that scenario, users could not withdraw their liquidity if they do not act quick enough.

To counter this case, the _reserveOps function was implemented which allows users to burn their ERC6909 tokens and then in turn reserves corresponding liquidity. This reserving mechanism decreases totalLiquidity such that the reserved amount effectively is unavailable to be withdrawn by other users and not allowed for minting options (usePosition).

Users can then withdraw this liquidity as soon as it comes available after the cooldown period has passed.

Appendix: Premium Distribution

Whenever an option is being minted, a premium is transferred from the user to the feeRecipient. This premium is then distributed off-chain to corresponding liquidity providers.

Appendix: Wildcard Mechanism

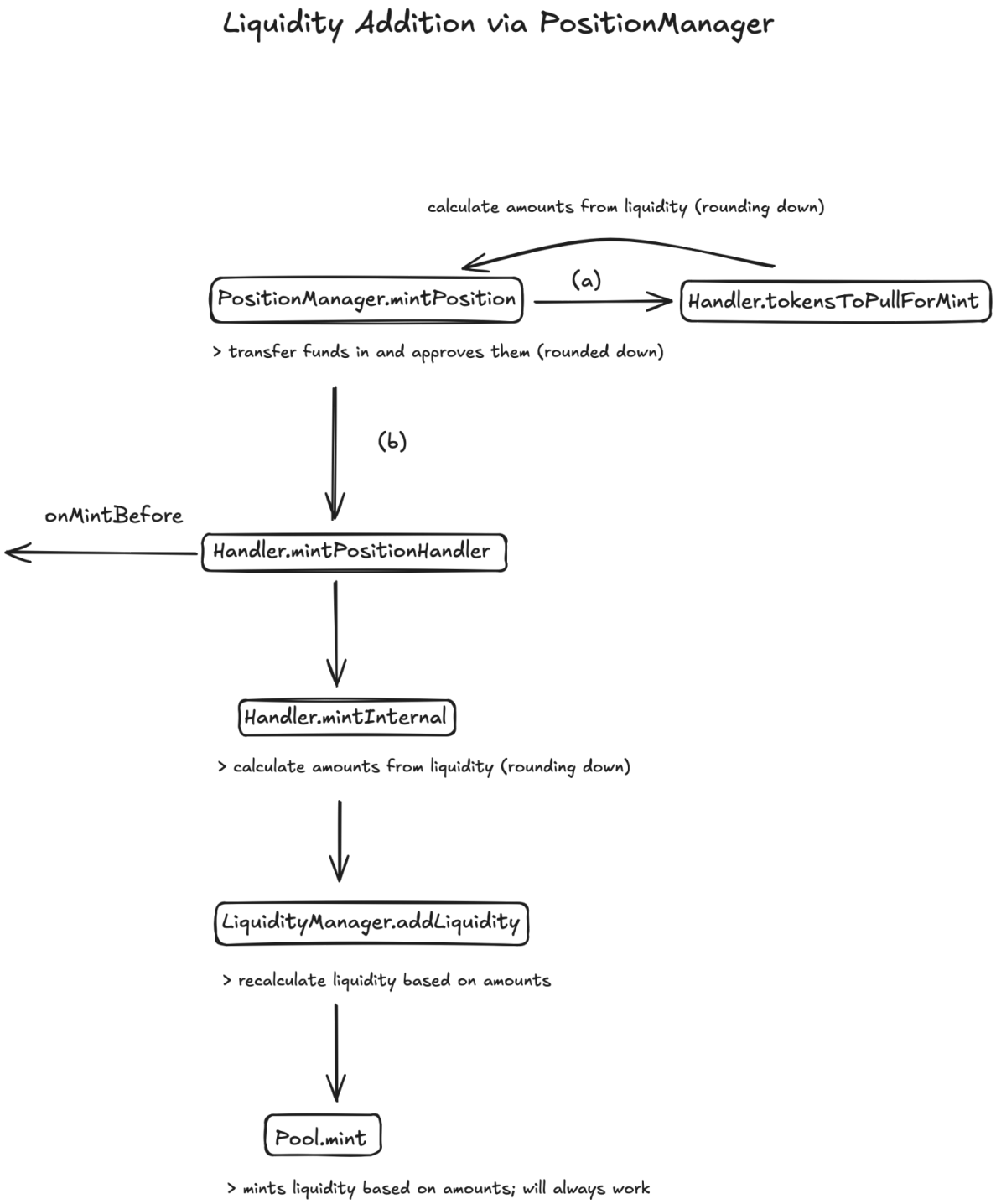
The wildcard mechanism allows for the following actions:

a) Reserving liquidity
b) Collecting fees
c) Splitting a position

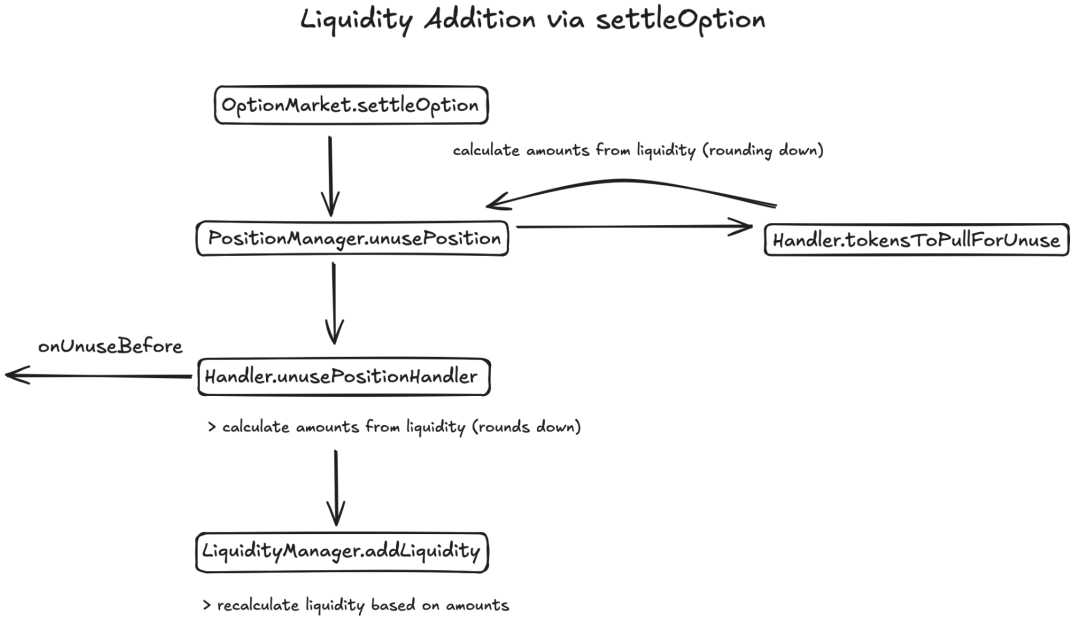
These are explained separately

Within the current scope, there are three distinct scenarios where liquidity is being added via the Handler:

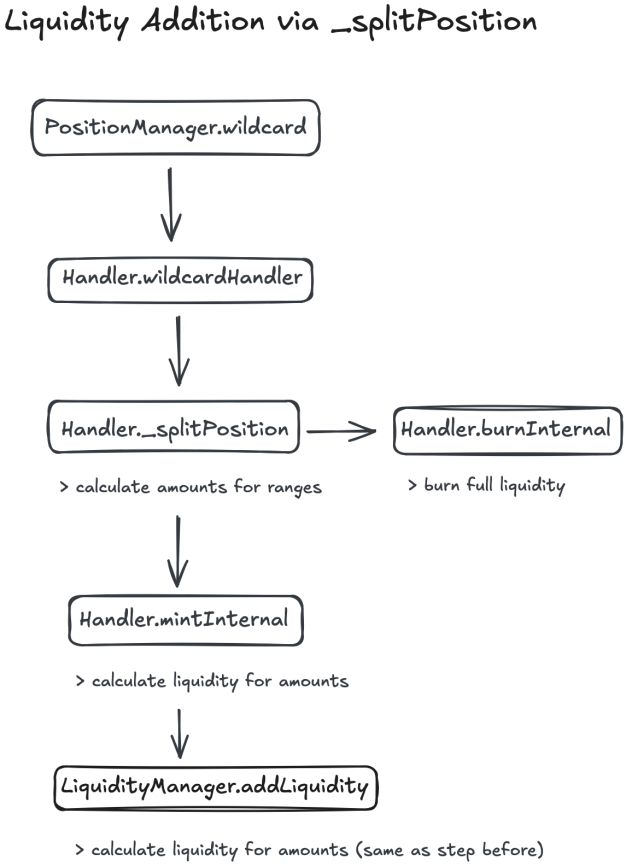
a) A user mints a position:



b) An option is settled:



c) A position is splitted:



Core Invariants:

INV 1: Free liquidity is determined as totalLiquidity - usedLiquidity

INV 2: _tokensToPull must return the amounts of tokens which are needed for a certain liquidity position

INV 3: mintPosition must mint tokenId ERC6909 to msg.sender, reflecting added liquidity

INV 4: usePositionHandler must return (tokens[], amounts[])

INV 5: Any collect call which is corresponding to liquidity removal must only ever collect anything from removed liquidity

INV 6: _feeCalculation must always be called before any liquidity change

INV 7: burnInternal cannot burn more than (totalLiquidity - liquidityUsed)

INV 8: tokensToPullForDonate must return tokenX or tokenY amount and never both

INV 9: tokensToPullForWildcard must return empty values

INV 10: _splitPosition must burn the full balance of a user

INV 11: _reserveOps with !isReserve can only be called after cooldown time has passed

INV 12: Within _reserveOps, !isReserved, liquidity to withdraw must never be larger than a users reserved liquidity

INV 13: _splitPosition must distribute new positions within the main position range

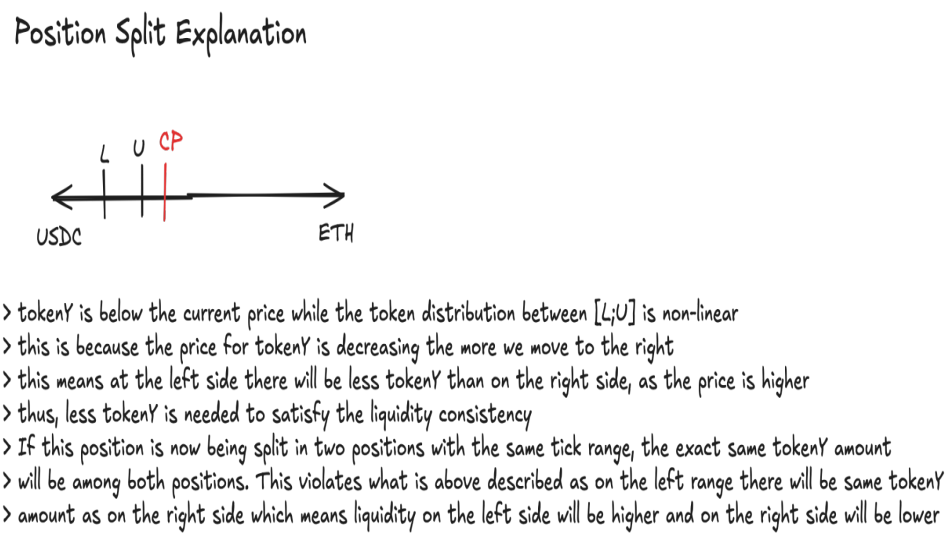
INV 14: reservedLiquidity can never be used for minting new options

INV 15: Within mintPositionHandler, it must always mint the exact same amount of ERC6090 tokens as to minted liquidity

Privileged Functions

- transferOwnership
- renounceOwnership
- updateHandlerSettings
- sweepTokens
- emergencyPause
- emergencyUnpause

Issue_31	Governance Issue: Full control over funds
Severity	Governance
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. - sweepTokens Furthermore, liquidity can be fully withdrawn by adding a whitelisted caller which can then call usePositionHandler and withdraw liquidity
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_32	Position splitting will alter liquidity
Severity	High
Description	The current way of splitting liquidity is calculating the average tokens per tick based on the overall tokens in the original range and the overall tick range. This is wrong as liquidity in UniswapV3 is not linear but changes with the curve:  The following can be used as numerical example: ## ## [0; 10000]; 100e18, curTick = 5000 > amountX = 17.22 > amountY = 28.40 If we now execute a swap in the range, it will always have liquidity of 100e18 But now if the position is being split, the liquidity for each range will be different. These different liquidites are the proof that the slippage will be changed by using this method. And if the slippage is changed this will lead to advanced exploitable states

0 = 792281625142643375935439503336
1 = 792321238233597991182869995568
5000 = 101729702841318637793976746270
7000 = 112428146980036109168897402316
10000 = 130621891405341611593710811006

We will now start with the first step of the exploit which is to swap down and provide tokenX while taking out tokenY and bringing the tick to 0

> inputX = 22.11
> outputY = 28.40

new state:

[0;10000]; curTick = 0
> amountX = 39345417784216524215
> amountY = 0
> liq = 100e18

getAmountsForLiq = 39345417784216524215

amountPerTick = 39345417784216524215 / 10000 = 3934541778421652

Now add 3934541778421652 * 5000 (19672708892108260000) to [0;5000]

[0;5000] = 19672708892108260000 (114e18)

6486800559311757 * 5000 (32434002796558785000) to [5000;10000]

[5000; 10000] = 88940525874653852074 (88.9e18)

	If we now execute a swap to the downside, this will be with liquidity of 88.9e18 instead of 100e18. The same input amount will have different slippage. This shows as clear proof that the linear liquidity split approach is incorrect. ## This is amplified for positions with a very large range, as the linear distribution will be in conflict with the price change from bottom to top of the range. This basically allows for altering the swap experience at will which can introduce swap-based exploits in case tokens are added at the incorrect tick due to price change pre-split. **Another important impact of the wrong splitting:** As splitting liquidity will deposit tokens equally instead of following the xyk curve, the actual liquidity value of the LP will decrease near the current price and increase far from the current price. This can be abused by other LPs to receive more swap fees as they can essentially reduce anyone's liquidity in the active ticks. This could prevent the initial LP from getting a lot of fees in a scenario where the price crabs for a long time.
Recommendations	We recommend completely removing the liquidity split logic. However, if desired to keep it: Consider fully refactoring the splitPosition function and making it only permissioned for the position owner. A full re-audit of the splitPosition function is required.
Comments / Resolution	Resolved, splitPosition has been completely removed.

Issue_33	Registration of used, unregistered hook allows for bricking positions permanently
Severity	**High**
Description	The registerHook function allows to assign different permissions to a hook and will mark a hook as registered: *function registerHook(address _hook, IHandler.HookPermInfo memory _info) external {* *if (hookRegistered[_hook]) {* *revert HookAlreadyRegistered();* *}* *hookPerms[_hook] = _info;* *hookRegistered[_hook] = true;* *}* This is exploitable in case a user has a position with an unregistered hook, as now anyone can register this hook with any permissions. This is due to the fact that there is currently no mandatory check that the provided hook is already registered when a position is being created. Therefore users can and will create positions with any address as a hook, such as address(0) for example, while this address is not registered yet. After position has been provided, a malicious user can simply register the hook with all flags to true which then reverts on all onBefore calls, since the hook address most likely does not support the expected function selector. Furthermore, a position with an unregistered hook can be registered with allowSplit = true which then allows splitting the position while this is unintended. This is amplified due to the fact that a hook address is mandatorily required when a position is being created.
Recommendations	Consider only allowing whitelisted hooks.

Comments / Resolution	Resolved, registerHook is now permissioned

Issue_34	Frontrun of registerHook call allows attacker to initialize hook incorrectly which can result im permanently locked funds
Severity	**High**
Description	The registerHook function can simply be called shortly before a position with an unregistered hook address is created (while the position creator has the intention to register the hook). This then in turn allows for either: a) DoS'ing usage of distinct hooks b) Tricking users into minting position with incorrectly configured hook This issue is different from the above mentioned issue as it targets to be created positions instead of already existing positions.
Recommendations	Consider only allowing whitelisted hooks.
Comments / Resolution	Resolved, registerHook is now permissioned.

Issue_35	Edge-case within isCurrentTickInRange during _splitPosition is not considered
Severity	Medium
Description	Currently, the isCurrentTickInRange check is as follows: *bool isCurrentTickInRange = cache.currentTick > spp.tickLower && cache.currentTick < spp.tickUpper;* This does not consider the situation where currentTick = lowerTick and lowerPrice < currentPrice which means the position is in range. In that scenario it will result in isCurrentTickInRange = false. It will therefore enter the following else-clause: *else {* *if (tL != cache.tLCurrentTick || tU != cache.tUCurrentTick) {* *revert InvalidTicks();* *}* *loopCache.amount0ForRange = (cache.initialAmount0 * cache.currentTickDiff0) / cache.ticksAtToken0;* *loopCache.amount1ForRange = (cache.initialAmount1 * cache.currentTickDiff1) / cache.ticksAtToken1;* *}* There are two outcomes: a) tl = tLCurrentTick & tU = tUCurrentTick Since isCurrentTickInRange = false, cache.currentTickDiff0 has never been set, thus it remains zero and therefore the product will always be zero. b) tl != tLCurrentTick or tU =! tUCurrentTick It will thus revert

	Both outcomes are erroneous and unexpected. Double check in notes later The same issue as in a) but less impactful is existing when a position is double sided as it now aims to distribute amounts proportionally to the tickRange position which covers currentTick but due to the linear distribution it will erroneously use incorrect amounts. UniswapV3 always calculates liquidity based on the smaller of both amounts: *else if (sqrtRatioX96 < sqrtRatioBX96) {* *uint128 liquidity0 = getLiquidityForAmount0(sqrtRatioX96, sqrtRatioBX96, amount0);* *uint128 liquidity1 = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioX96, amount1);* *liquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;* (This approach could indeed work if the range is geometrically around the price of 1 which is however in 99.9% never the case) This leads to a situation where one of both amounts is not fully consumed and thus refunded to the user. This issue can have potentially other, unexpected impacts.
Recommendations	We recommend completely removing the liquidity split logic. However, if desired to keep it: Consider fully refactoring the splitPosition function and making it only permissioned for the position owner. A full re-audit of the splitPosition function is required.
Comments / Resolution	Resolved, splitPosition has been completely removed.

Issue_36	Double sided split calculation will always revert when currentTick is a multiple of tickSpacing
Severity	Medium
Description	As already mentioned, the splitting approach using amounts in itself is already incorrect, as tokens are not distributed linearly among a range. On top of that, in the scenario where the currentTick is a multiple of tickSpacing and that the position is double sided, the position must be split in a small one tickSpacing wide position in order to make it easier to calculate the amount to deposit in the other section. So for example, if the tickSpacing is 10 and the currentTick is 55. For a position going from 0 to 100, the split array must be in the form of [0, ..., 50, 60, ..., 100]. In the 0...50 and 60...100 parts, the liquidity will be single sided and can be any multiple of tickSpacing and in 50,60 it will be double sided. However, in the event of the currentTick being a multiple of tickSpacing the function will always revert due to the requirement cache.currentTick > tU || cache.currentTick < tL. Using the previous example, but the currentTick is at 60, if we assume that the split array will be [0, 50, 60, 100], the last split, with tL = 60 and tU = 100, will revert because the check cache.currentTick > tU || cache.currentTick < tL returns false it will go in the else case: *} else {* *if (tL != cache.tLCurrentTick || tU != cache.tUCurrentTick) {* *revert InvalidTicks();* *}* *....* *}* The check will always revert because tL and tU can't be equal to the cached tL and tU.

	However, if the pair uses a tickSpacing of 1, it means that no double sided position can be split as it will always revert.
Recommendations	We recommend completely removing the liquidity split logic. However, if desired to keep it: Consider fully refactoring the splitPosition function and making it only permissioned for the position owner. A full re-audit of the splitPosition function is required.
Comments / Resolution	Resolved, splitPosition has been completely removed.

Issue_37	ticksAtToken0/1 calculation is incorrect
Severity	**Medium**
Description	The _splitPosition function determines how much tokens to linearly distribute by simply calculating ticksAtToken0/1 as follows: *cache.ticksAtToken0 = _calculateTickDifference(cache.currentTick, spp.tickSplits[spp.tickSplits.length - 1]);* *cache.ticksAtToken1 = _calculateTickDifference(cache.currentTick, spp.tickSplits[0]);* This is not correct as the currentTick cannot be used as a point of reference for the ratio calculation of tokenX and tokenY. In the scenario where currentPrice > priceFromCurrentTick, this calculation will be incorrect as it still uses currentTick for boundary purposes, while in fact between currentTick and currentPrice tokenY will be existing and only >= currentPrice tokenX will be existing.
Recommendations	We recommend completely removing the liquidity split logic. However, if desired to keep it: Consider fully refactoring the splitPosition function and making it only permissioned for the position owner. A full re-audit of the splitPosition function is required.
Comments / Resolution	Resolved, splitPosition has been completely removed.

Issue_38	Gas-griefing attack via custom hooks
Severity	**Medium**
Description	It is possible to mint a position with custom hooks. While there is an obvious issue with malicious hooks which prevent certain executions such as settlements, there is a less obvious risk with gas-griefing which can actually trick users into paying more gas than expected with a profit for the position owner. This could be a very trivial call to 1Inch's "CHI" implementation which then forces the caller to mint gas tokens to the position owner. This can be implemented subtle such that no one actually notices it.
Recommendations	Consider only allowing whitelisted hooks.
Comments / Resolution	Resolved, it is now only possible to use whitelisted hooks in the Options system.

Issue_39	Cooldown period is not stored upon reserving
Severity	**Medium**
Description	Reserved liquidity can only be withdrawn once the cooldown time has been reached. This is based on the time when liquidity was reserved and the current cooldown time: *if (rld.lastReserve + reserveCooldownHook[_params.hook] > block.timestamp) revert BeforeReserveCooldown();* An issue arises if the cooldown period is changed after a user has reserved liquidity, as it now violates user assumptions. This means if a user executes _reserveOPs while the cooldown period is 86400 for the corresponding hook, this value will not be stored. Instead if governance changes it to 604800, the user will have to wait until 604800 seconds are passed, while initially assuming he only needs to wait for 86400 seconds.
Recommendations	Consider caching the initial cooldown period. It has to be noted that a subsequent call with isReserve = true would then allow for overriding it, so there is no non-intrusive fix for this issue which leads us to the recommendation to simply **acknowledge it and communicate each change with the community beforehand**.
Comments / Resolution	Acknowledged. Such changes will be communicated with users beforehand.

Issue_40	Unconsumed approvals from PositionManager can be added around wrong tick by using onMintBefore hook
Severity	**Low**
Description	The PositionManager transfers tokens in and approves them in the first instance. Due to the logic of how liquidity is added it will often result in some dust remaining in the PositionManager and not being added as liquidity (we have added a separate issue for this). In case of low decimal tokens this can become more significant. These funds can be added as liquidity and bought out at the wrong price via sophisticated pre and post liquidity addition swap. The exploiter first calls the mintPosition function which transfers funds in and approves this to the Handler contract. Then during the onMintBefore hook, the exploiter executes a swap which alters the tick. The liquidity which is added will now partially consist out of the unconsumed tokens at the wrong price which can be retrieved via swapping back for a profit.
Recommendations	Consider allowing only for whitelisted hooks.
Comments / Resolution	Acknowledged, it has to be noted while positions with arbitrary hooks can still be created, these cannot be used in the Options system.

Issue_41	reserveCooldownHook check can be bypassed for new position
Severity	**Low**
Description	Currently, the cooldown period is bound to each hook: *mapping(address => uint64) public reserveCooldownHook;* Therefore, if a position with a new hook is being created, it has an initial cooldown period of zero, which allows for immediate execution of a reserved position. This is due to the fact that governance first needs to explicitly set the cooldown period for a hook
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_42	DoS of splittable position
Severity	**Low**
Description	Currently, it is possible to split positions permissionless if they have the hook enabled for this. This opens up a DoS vector by splitting one range into hundreds/thousands of mini positions which then requires the original position owner to burn all these separately. This will furthermore introduce complexity when collecting fees from the newly created positions.
Recommendations	We recommend completely removing the liquidity split logic. However, if desired to keep it: Consider fully refactoring the splitPosition function and making it only permissioned for the position owner. A full re-audit of the splitPosition function is required.
Comments / Resolution	Resolved, splitPosition has been completely removed.

Issue_43	Wildcard handler executions lack tokenId validation
Severity	**Low**
Description	All wildcardHandler executions can currently be done with an invalid tokenId. While we could not find a specific exploit due to this, most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider ensuring that a tokenId is indeed existing, this could be done by simply checking that the supply is non-zero.
Comments / Resolution	Acknowledged.

Issue_44	Truncation within multiple spots in _splitPosition
Severity	**Low**
Description	The _splitPosition function executes multiple arithmetic operations, including the following: *(cache.userAmount0 * cache.currentTickDiff0) / cache.ticksAtToken0,* *(cache.userAmount1 * cache.currentTickDiff1) / cache.ticksAtToken1* *cache.amount0PerTick = cache.userAmount0 / cache.ticksAtToken0;* *cache.amount1PerTick = cache.userAmount1 / cache.ticksAtToken1;* These can result in a truncation to zero in case tokens have low decimals while the range is large. This can be abused to simply remove liquidity and transferring funds to the owner.
Recommendations	We recommend completely removing the liquidity split logic. However, if desired to keep it: Consider fully refactoring the splitPosition function and making it only permissioned for the position owner and increasing precision for intermediate divisions. A full re-audit of the splitPosition function is required.
Comments / Resolution	Resolved, splitPosition has been completely removed.

Issue_45	Increased totalLiquidity can never be withdrawn
Severity	**Informational**
Description	Within the unusePositionHandler function, the following edge-case condition is existent: *else {* *tki.totalLiquidity += (liquidity - tki.liquidityUsed);* *tki.liquidityUsed = 0;* *}* This will essentially increase totalLiquidity in case the liquidity addition is larger than what was used during the option minting. If that happens, the delta remains always unclaimable as there are no ERC6909 tokens to redeem for.
Recommendations	Consider acknowledging this issue, as it is only about dust and eventual changes can introduce further side-effects.
Comments / Resolution	Acknowledged.

Issue_46	Violation of CEI within _collectFees
Severity	**Informational**
Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes): *IRamsesV3Pool(address(_pool)).collect(* *feeReceiver, uint256(0), _tickLower, _tickUpper,* *tki.tokensOwed0, tki.tokensOwed1* *);* *tki.tokensOwed0 = 0;* *tki.tokensOwed1 = 0;*
Recommendations	Consider acknowledging this issue. In the scenario where it is desired to be fixed, consider casting the original amount beforehand and using it as parameter for the collect function, as otherwise funds would remain stuck.
Comments / Resolution	Acknowledged.

Handlers/shadow-v3

LiquidityManager

The LiquidityManager contract is responsible for the direct interaction with the UniswapV3Pool (RAMSES) for liquidity addition and is inherited as part of the Handler. There are two distinct liquidity addition scenarios:

a) Adding liquidity while transferring in funds from the caller address. This is happening during the position creation and during option settlement. Funds are being pulled from the PositionManager

b) Adding liquidity while funds are already sitting within the Handler contract. This is happening whenever liquidity is being split into smaller positions.

For callback validation purposes, the contract uses the same pattern as the UniswapV3 NonfungiblePositionManager:

https://github.com/Uniswap/v3-periphery/blob/main/contracts/base/LiquidityManagement.sol#L25

https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/CallbackValidation.sol#L28C1-L35C6

Core Invariants:

INV 1: Only a valid UniswapV3Pool can call uniswapV3MintCallback

Privileged Functions

- none

Issue_47	Insufficient liquidity is minted during most cases
Severity	**Low**
Description	The addLiquidity function is invoked on the following occasions: a) Users create a new position b) An option is settled c) A position is split The callflows for these scenarios are illustrated within “Appendix: Liquidity Addition” This is based on the fact that liquidity is re-calculated based on the fetched amounts from the desired liquidity. Therefore, the minted liquidity can be slightly lower than expected. The current step for adding liquidity as user is as follows (example): 1. Provide desired liquidity amount as parameter 2. Calculate amounts for desired liquidity amount using library 3. Transfer required amounts in from user 4. Recalculate amounts from liquidity parameter 5. Recalculate liquidity from amounts 6. Transfer required amounts from PositionManager to UniswapV3Pool for liquidity During this process, the final liquidity which is recalculated in step 5 will most of the time be smaller than the initially provided liquidity parameter.
Recommendations	Consider adjusting the liquidity addition flow to follow a similar mechanism as UniswapV3’s NFPM. *The refactored liquidity addition flow must be fully re-audited which falls outside of the resolution scope.*

Comments / Resolution	Acknowledged.

Issue_48	zkSync deployment doesnt work
Severity	**Low**
Description	Currently, the pool is deterministically crafts the pool address in the usual know scheme: *function computeAddress(address _factory, PoolKey memory key) internal view returns (address pool) {* *require(key.token0 < key.token1);* *pool = address(* *uint160(* *uint256(* *keccak256(* *abi.encodePacked(* *hex"ff",* *factory,* *keccak256(abi.encode(key.token0, key.token1, key.tickSpacing)),* *POOL_INIT_CODE_HASH* *)* *)* *)* *)* *);* *}* This does not work for zkSync, the determination mechanism there is different: https://foundry-book.zksync.io/tutorials/create2-tutorial
Recommendations	Consider crafting a special LiqudiityManager (LiquidityManageZkSync) for this purpose.
Comments / Resolution	Acknowledged.

ShadowV3Handler

The ShadowV3Handler contract is responsible for determining the correct liquidity addition flow and direct interactions with Shadow to remove liquidity and collect Gauge rewards. It handles whether liquidity is being added as self call (during liquidity split) or as normal call (during settlement and position creation).

Privileged Functions

- transferOwnership
- renounceOwnership
- getGaugeRewards

Issue_49	The getGaugeRewards exposes a index parameter while it is always set to 0
Severity	**Informational**
Description	Currently, the getGaugeRewards function exposes the index parameter even if it is always set to 0 within the codebase. This is misleading and unnecessary and could make it harder for users to query their positions.
Recommendations	Consider removing the index parameter and always use 0 as long as the codebase doesn't use any other index.
Comments / Resolution	Acknowledged.

Handlers/Hooks

BoundedTTLHook_0Day

The BoundedTTLHook_0Day contract is a simple template hook implementation which only allows for minting options if the expiry date is > 24 hours from the current block timestamp. It furthermore allows for whitelisting handlers and registering the hook on handlers.

Core Invariants:

INV 1: Only whitelisted handlers can call onPositionUseBefore

INV 2: If an option expires in < 24 hours, the onPositionUseBefore function reverts.

Privileged Functions

- transferOwnership
- renounceOwnership
- registerHook
- updateWhitelistedAppsStatus

Issue_50	Hook can be griefed because anyone can register the hook directly
Severity	**Medium**
Description	Currently it is possible that anyone calls the registerHook function on the Handler contract. This would allow for trivial griefing by simply using the valid hook address and setting it up with invalid parameters. This issue is existing for the BoundedTTLHook_1Week contract as well.
Recommendations	Consider making the registerHook function on the Handler permissioned and only allowing for whitelisted hooks.
Comments / Resolution	Resolved, the registerHook function is now permissioned.

Issue_51	Forced split allowance
Severity	**Low**
Description	The current hook is being registered with allowSplit: true. This is a subtle property which must be noted by users using this hook. This issue is existing for the BoundedTTLHook_1Week contract as well.
Recommendations	Consider implementing NATSPEC which clearly highlights this property.
Comments / Resolution	Acknowledged.

BoundedTTLHook_1Week

The BoundedTTLHook_1Week contract is similar to the BoundedTTLHook_0Day contract with the only difference that expiry for new options must not be > 1 week in the future.

Core Invariants:

- same as above

Privileged Functions

- same as above

Same issues found as above.

Periphery

OpenSettlement

The OpenSettlement contract will be marked as a whitelisted settler on the OptionMarketOTMFE contract which is thus allowed to settle expired options. Contrary to the scenario where a whitelisted EOA settles an option, will the OpenSettlement contract distribute the profit to the rightful option holder, while with an EOA settlement, the caller will simply keep the profit and the option owner will not receive anything.

Core Invariants:

INV 1: Only expired options can be settled

INV 2: Settling must distribute profit to option owner

Privileged Functions

- transferOwnership
- renounceOwnership
- setSettleFeeProtocol
- setSettleFeePublic
- setIsWhitelistedSettler
- setPublicFeeRecipient

Issue_52	Profit can be stolen due to lack of optionId validation
Severity	**High**
Description	The optionId input parameter is used to determine the profit recipient by fetching the owner of the option: *function openSettle(* *IOptionMarketOTMFE market,* ***uint256 optionId,*** *IOptionMarketOTMFE.SettleOptionParams memory settleParams* *) external returns (IOptionMarketOTMFE.AssetsCache memory) {* *ac.assetToGet.safeTransfer(market.ownerOf(optionId), ac.totalProfit);* This optionId is however never validated which means literally any ITM optionId can be settled without ensuring that it is related to the provided optionId. This means an attacker can settle any option while stealing the profit.
Recommendations	Consider deriving optionId from settleParams, ensuring that the original option owner receives the profit.
Comments / Resolution	Acknowledged. The client indicated that this contract will not be used in production.

Issue_53	Underflow in feeProtocol calculation can result in DoS
Severity	**High**
Description	Whenever the caller is not a whitelistedSettler, the openSettle function will calculate feeProtocol and feeCaller as follows: *feeProtocol = (ac.totalProfit * (settleFeeProtocol - settleFeePublic)) / SETTLE_FEE_PRECISION;* *feePublic = (ac.totalProfit * settleFeePublic) / SETTLE_FEE_PRECISION;* In the scenario where settleFeeProtocol < settleFeePublic, the calculation reverts due to underflow and settlement can never happen until governance adjusts fees. This is specifically important because it might be a legitimate scenario to set settleFeePublic > settleFeeProtocol, for example in cases where settlement via a whitelisted settler should only take a certain fee percentage for the protocol while in cases without a whitelisted seller, the public fee should be higher than the protocol fee in the previous described scenario. For example the protocol fee should be 5% when settling via whitelistedSettler and the public fee should be 10% when settling via non-whitelisted settler. Furthermore, there is no validation at all for the nominal fee percentages.
Recommendations	Consider ensuring that settleFeeProtocol is always larger or equal to settleFeePublic as well as implementing reasonable upper limitations for fee settings in general. Furthermore, consider ensuring that publicFeeRecipient cannot be address(0).
Comments / Resolution	Acknowledged. The client indicated that this contract will not be used in production.

Issue_54	Open settlement is incompatible with profitable call-path
Severity	**Medium**
Description	The settleOption function incorporates multiple different callpaths, including one where an option is not fully ITM but partially: *if (isAmount0 && ac.isSettle == true) {* *ac.assetToUse.approve(address(positionManager), amount0);* *ac.assetToGet.approve(address(positionManager), amount1);* *uint256 actualAmount0 = LiquidityAmounts.getAmount0ForLiquidity(* *opTick.tickLower.getSqrtRatioAtTick(),* *opTick.tickUpper.getSqrtRatioAtTick(),* *uint128(liquidityToSettle)* *);* *ac.assetToGet.transferFrom(msg.sender, address(this), amount1);* *ac.assetToUse.transfer(msg.sender, actualAmount0 - amount0);* *}* The openSettle function is simply not compatible with this callpath as it will never store any tokens nor approve any tokens to the OptionMarketOTMFE contract.
Recommendations	Since one important recommendation within the OptionMarketOTMFE contract is to ensure permissioned settlements, this contract should not be used at all. However, if intended to be used in the future under other circumstances, consider refactoring the settlement logic to allow for compatibility with partial ITM callpath. This requires a re-audit of the openSettle function.

Comments / Resolution	Acknowledged. The client indicated that this contract will not be used in production.

Issue_55	Idle funds within OpenSettlement contract can be taken out
Severity	**Low**
Description	The openSettle function allows for providing an arbitrary market address. This can be abused to provide a malicious market address which returns profit that was not actually received, allowing for stealing idle assets in the contract. This has only been rated as low severity because the OpenSettlement contract is not meant to hold any funds.
Recommendations	Consider validating the market parameter with a whitelist.
Comments / Resolution	Acknowledged.

Issue_56	openSettle will never work if the option holder is blacklisted for profit token
Severity	**Low**
Description	In the scenario where the original option holder is blacklisted for the profit token, the openSettle function will never work because the transfer always reverts.
Recommendations	Consider settling an option directly on the OptionMarketOTMFE contract in this scenario.
Comments / Resolution	Acknowledged.

Periphery/Routers

MultiswapRouter

The MultiswapRouter is a simple router implementation with the following features:

- Wrap ETH to WETH
- Interact with arbitrary swapper contracts
- Mint options

The contract is completely permissionless.

Important: The MultiswapRouter is not meant to hold any funds and the sweep function must be called at the end of every execution.

Core Invariants:

INV 1: All remaining funds must be swept after execution

Privileged Functions

- none

Issue_57	Lack of whitelisting for multiple parameters
Severity	**Low**
Description	Currently, no parameters are validated. This includes swappers, WETH, markets etc. While we could not find an exploit, most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic. The attack surface is limited to malicious approval consumption, since the contract is not intended to hold funds.
Recommendations	Consider implementing a whitelist for different parameters.
Comments / Resolution	Acknowledged.

Issue_58	Lack of slippage check within swap
Severity	**Low**
Description	The swap functions allows the interaction with different swappers. Currently there is no slippage check which ensures that the output amount meets a desired threshold. Thus, it is assumed that this will be handled in the swapper contract.
Recommendations	Consider only using swapper contracts with incorporated slippage check.
Comments / Resolution	Acknowledged.

Swapper

OnSwapReceiver

The OnSwapReceiver contract is a swapper contract which expects for the input token to be received as balance and then executes an arbitrary call with the target to receive a specific amount of output tokens.

Core Invariants:

INV 1: _amountIn must be transferred before onSwapReceived is called

INV 2: After the external call, the contract must at least own minAmountOut of tokenOut

Privileged Functions

- none

Issue_59	Arbitrary execution via external call
Severity	**Medium**
Description	Currently it is possible to provide any address with any data which then allows users to execute arbitrary calls. This can become problematic in case the contract is used as an approved swapper within the Option contract. It furthermore allows for stealing all approvals which are made to this contract.
Recommendations	Consider never using this contract with the Option contract.
Comments / Resolution	Acknowledged. The client indicated that only the native primePool will be used for swaps.

Issue_60	Usage of exactOut does not refund input amount
Severity	**Low**
Description	Currently, any call can be executed. If a swap is being executed using the exactOut methodology, no refund of the potential leftover input token is happening.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_61	Idle tokens can be withdrawn
Severity	**Low**
Description	The contract explicitly expects to receive the output token after the swap as it then executes a transfer of the same: *if (amountOut < minAmountout) {* *revert OnSwapReceiver__InsufficientAmountOut();* *}* *tokenOut.transfer(msg.sender, amountOut);* This opens the path to simply withdraw any idle tokens by calling a custom to adress which doesnt actually transfer the output token back to the OnSwapReceiver contract. This then transfers out idle tokens to the caller. This issue is only marked as low severity since the contract does not intend to hold any tokens.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_62	Pre-existing balance flows into output
Severity	**Informational**
Description	Currently, there is no before-after check to gauge how much has been received through the external call. Therefore, also previous existing funds will flow into the minAmountOut check, which can falsify the swap expectation.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

OneInchSwapper

The OneInchSwapper contract is a swapper contract which expects for the input token to be received as balance and then executes a call to the 1Inch aggregation router with arbitrary swap data.

Core Invariants:

INV 1: The input token must be received before onSwapReceived is called.

Privileged Functions

- none

Issue_63	Usage of malicious calldata allows for routing through arbitrary pairs and custom executor addresses
Severity	Medium
Description	The _swapData parameter is completely unvalidated, which means users can pass any arbitrary data. This essentially allows for custom execution in case it is routed through a malicious token or towards a custom caller address (which is standard for 1Inch swaps): *function _execute(* *IAggregationExecutor **executor**,* *address srcTokenOwner,* *uint256 inputAmount,* *bytes calldata data* *) private returns(uint256 result) {* *bytes4 executeSelector = executor.execute.selector;* *assembly ("memory-safe") { // solhint-disable-line no-inline-assembly* *let ptr := mload(0x40)* *mstore(ptr, executeSelector)* *mstore(add(ptr, 0x04), srcTokenOwner)* *calldatacopy(add(ptr, 0x24), data.offset, data.length)* *mstore(add(add(ptr, 0x24), data.length), inputAmount)* *if iszero(call(gas(), **executor**, callvalue(), ptr, add(0x44, data.length), 0, 0x20)) {* *returndatacopy(ptr, 0, returndatasize())* *revert(ptr, returndatasize())* *}* *result := mload(0)* *}* *}*
Recommendations	We simply recommend only using the native primePool. In the case where this contract is ever planned to be used, a full integration

	audit with the 1Inch router must be conducted which uncovers all potential arbitrary call executions and edge-cases.
Comments / Resolution	Acknowledged. The client indicated that only the native primePool will be used for swaps.

Issue_64	Idle tokens can be withdrawn
Severity	**Low**
Description	The contract approves the input token to the oneInchRouter without actually transferring it in. This is the correct business logic flow based on the Options contract as funds are transferred in via a simple transfer. This opens a path to withdraw any tokens which are sitting idle in the contract as anyone can simply call the onSwapReceived function with the exact idle amount as _amountIn which then allows for taking it out if the “to” address is a custom contract which executes a transferFrom. This issue is only marked as low severity since the contract does not intend to hold any tokens.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_65	Lack of return value usage
Severity	**Informational**
Description	The onSwapReceived function returns a uint256: *function onSwapReceived(address _tokenIn, address _tokenOut, uint256 _amountIn, bytes memory _swapData)* *external* *returns (uint256 amountOut)* *{* It does however never determine a return value which means it will always be zero.
Recommendations	Consider either removing the return value or declaring it.
Comments / Resolution	Acknowledged.

SwapRouterSwapper

The SwapRouterSwapper contract is a swapper contract which expects for the input token to be received as balance and then executes a exactInputSingle call to the shadow router which then executes a swap on the primePool (or corresponding pools with a different fee) of the corresponding Option contract.

Core Invariants:

INV 1: The input token must be received before onSwapReceived is called.

Privileged Functions

- none

Issue_66	Lack of fee validation increases user flexibility
Severity	**Low**
Description	Currently, any fee tier for the corresponding tokenIn and tokenOut pool can be used. This will greatly increase user flexibility as in certain invalid states, a user could manipulate a low liquidity pool/new pool to a certain tick and liquidity just to then successfully execute a trade which should not be possible based on the invalid state. While the previous exploit could have potentially been prevented by only allowing for this swapper contract, the additional fee-tier flexibility could have allowed the exploiter to still execute the swap.
Recommendations	Consider if it makes sense to validate the fee to ensure swaps can only be executed via the primePool of the Option contract.
Comments / Resolution	Acknowledged.

Issue_67	Idle tokens can be withdrawn
Severity	**Low**
Description	The contract approves the input token to the sr without actually transferring it in. This is the correct business logic flow based on the Options contract as funds are transferred in via a simple transfer. This opens a path to withdraw any tokens which are sitting idle in the contract as anyone can simply call the onSwapReceived function with the exact idle amount as _amountIn which then allows for taking it out via a simple swap to another token. This issue is only marked as low severity since the contract does not intend to hold any tokens.
Recommendations	Consider acknowledging this issue.

Comments / Resolution	Acknowledged.

Issue_68	Swapper execution can be sandwich-attacked
Severity	**Informational**
Description	The contract intends to route through one specific pool and sells tokenX to tokenY or vice-versa. A sandwich-attack around this swap can be used to first sell tokenX which drives the price lower such that the user's range still remains single-sided after the swap and the liquidity provision does not revert, followed by a re-purchase of tokenX for a lower price. The sandwich-attacker essentially gained a nominal amount of tokenX. (or vice-versa) This issue has only been rated as informational severity since a minAmountOut can be provided. However, if no very strict minAmountOut is used, the swap can be ambiguous, manipulated by the sandwich-attacker.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Base

PositionManager

The PositionManager is the intermediate contract between the OptionMarketOTMFE and the Handler contract and it furthermore exposes entry functions for users to create positions (mint liquidity) and remove positions (burn liquidity).

All call-flows which are handler related will flow through this contract. These are the following:

- mintPosition: Allows users to create a new position
- burnPosition: Allows users to remove an existing position
- usePosition: Is called by the OptionsMarketOTMFe contract whenever an option for a position is minted
- unusePosition: Is called by the OptionsMarketOTMFe contract whenever an option is settled
- donateToPosition: Handles the premium transfer
- wildcard: Allows for splitting, reserving and collecting fees from positions

The contract uses a specific modifier pattern (onlyWhitelistedHandlersWithApps) to restrict Market <-> Handler interactions. It basically ties markets to handlers.

The current architecture has only been audited in relation to:

OptionMarketOTMFE <-> PositionManager <-> Shadow Handler

Appendix: Premium Distribution

Core Invariants:

INV 1: All modifier settings must be appropriately set

Privileged Functions

- transferOwnership
- renounceOwnership
- sweepTokens
- updateWhitelistHandlerWithApp
- updateWhitelistHandler

Issue_69	Governance Issue: Full control over funds
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, governance can set whitelistedHandlersWithApp to own address with corresponding handler and remove liquidity via usePosition.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_70	Position minting lacks slippage check
Severity	High
Description	Currently, users can mint a new position by providing the desired liquidity amount as a parameter within the PositionManager.mintPosition function. From the liquidity parameter it will then calculate the required amounts to be transferred in. This is contrary to the UniswapV3 logic and wrong, as any change in price will completely change the ratio in which tokens will be added. If for example a user targets to add liquidity around [-1000; 1000] while currentTick = 0, it will require the user to provide the same tokenX and tokenY amount around the tick = 0 which is price of 1. If now the price is changed which can either be on purpose with a sandwich attack or simply by coincidence, this will change the ratio of tokens to be provided and they will also be added at a wrong tick. To illustrate an extreme example where a sandwich-attacker swaps the tick to 1000, it will then transfer only tokenY in between [-1000; 1000] while the real price is tick 0 (price = 1). This means tokenY will be provided at a completely wrong price, allowing the sandwich-attacker to swap back and essentially buying tokenY from the liquidity provider at a very cheap price. The larger the range, the worse the impact and it will always result in a loss of funds for the position creator as liquidity will be added at the wrong tick (even though the desired range is always guaranteed). For example, if the user wishes to deposit 100e18 liquidity, it should take 4.876 token0 and 4.876 token1. However if the liquidity add is frontran to move the current tick to 1000, it will deposit 10 token1 that can be swapped back by the attacker to receive 5.124 token1 for only 4.876 token0, which is a profit of ~$0.25. If the position was from -100_000 to 100_000, the user should have deposited 99.38 of both tokens, but will end up depositing over 14836 token1, which can be swapped back for a profit of ~$14638 for the attacker. As said the bigger the position and the liquidity amount, the worse the attack

	It is notable that a slippage check is irrelevant during burn, as a price change or frontrun during burn always is in favor of the pair which means that the user will always receive more nominal value than expected.
Recommendations	The liquidity addition flow must be rewritten to follow the Uniswap NonfungiblePositionManager flow: a) Determine desired amounts and minimum amounts b) Calculate liquidity from these amounts c) Call UniswapV3Pool.mint with the calculated liquidity. d) Execute a slippage check https://github.com/Uniswap/v3-periphery/blob/main/contracts/NonfungiblePositionManager.sol#L141 https://github.com/Uniswap/v3-periphery/blob/main/contracts/base/LiquidityManagement.sol#L71 It is important to understand that this will require rewriting the cross-contract logic between PositionManager and the Handler and it must be ensured that no tokens will then be stuck in the PositionManager contract. The new liquidity addition flow must then be fully audited.
Comments / Resolution	Acknowledged. The client will only allow adding liquidity via a router which is implemented by the frontend. The router will have such a slippage check. Users are thus effectively not vulnerable when adding liquidity. We recommend validating the correctness of the router implementation.

Issue_71	Insufficient pool address validation within mintPosition
Severity	Low
Description	The mintPosition function allows a user to provide a pool parameter which is then consulted for price, token0 and token1. This parameter is not validated in the first instance which thus allows a user to trick the amounts return value: *(address[] memory tokens, uint256[] memory amounts) = _handler.tokensToPullForMint(_mintPositionData);* Later in the callpath, when liquidity is being added, the pool is then derived from the key which makes sure it is a pool deployed by the UniswapV3Factory: *PoolKey memory poolKey = PoolKey({token0: params.token0, token1: params.token1, tickSpacing: params.tickSpacing});* *pool = IUniswapV3Pool(computeAddress(factory, poolKey));* Therefore, it is possible to force a deviation between the initially provided pool address and the pool where liquidity is added. As already mentioned, it allows a user to manipulate the required and approved amounts. In some scenarios this can lead to the consumption of unconsumed approvals and dust in the PositionManager due to the real pool requiring different amounts. Furthermore, this can result in stuck funds as now the liquidity removal attempts to call the incorrect pool which does not have any liquidity at all. **Fortunately, the Options contract implements a whitelisting mechanism for pools when minting options, this prevents a sophisticated insolvency attack, which can be described as follows (but is prevented; the attack description is solely for informational purpose):**

While liquidity will always be added to the factory determined pool, the actual pool address which will be stored to the position is the parameter provided address:

tokenId: uint256(
*keccak256(abi.encode(address(this), **_params.pool,***
_params.hook, _params.tickLower, _params.tickUpper))
),

This will then result in calling this malicious pool during liquidity removal, while liquidity has been added to the correct pool:

(uint256 amount0, uint256 amount1) =
_removeLiquidity(_params.pool, _params.tickLower,
_params.tickUpper, _params.liquidityToUse);

*IRamsesV3Pool(address(**_params.pool***)).collect(*
msg.sender, uint256(0), _params.tickLower,
_params.tickUpper, uint128(amount0), uint128(amount1)
);

(amount0, amount1) =
*IRamsesV3Pool(address(**_pool***)).burn(uint256(0), tickLower,*
tickUpper, liquidity);

* Custom/Malicious pool address

During mintOption it will then attempt to remove liquidity from the malicious pool which can simply return with the desired parameters. The Options contract then assumes that these tokens have been received - they however were not received.

*This step is prevented due to the approvedPools validation

If the user calls the settleOption function, the wrong pool will be consulted which can return any price such that the option is ITM or

	OTM. The end result will always be that liquidity is being re-added to the real pool via the _addLiquidity function. This flow effectively allows a malicious user to make the contract insolvent. It has to be noted that the attacker himself will also lose funds by executing this strategy as funds are effectively donated to the “real” pool which can never be withdrawn because no corresponding ERC6909 token is existing. Furthermore, it would also allow for manipulating the _feeCalculation logic as now a malicious pool can return an incorrect: *(, uint256 feeGrowthInside0LastX128, uint256 feeGrowthInside1LastX128,,) = _pool.positions(positionKey);* which then in turn increases tokensOwed0/1 incorrectly. However, there is no harm with that as it will then consult the incorrect pool address during the collect call.
Recommendations	Consider implementing a whitelist for allowed pools within the PositionManager which is settable by governance, similar to the approvedPools mapping within the mintOption function.
Comments / Resolution	Acknowledged.

Issue_72	No minimum liquidity threshold within mintPositionHandler
Severity	**Low**
Description	Currently, it is possible to create a position with any liquidity size, including a very low value. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider implementing a threshold.
Comments / Resolution	Acknowledged. It has to be mentioned that such a limitation has indeed been incorporated within the Options contract itself during the mintOption function.

Issue_73	Full position burning can be griefed
Severity	**Low**
Description	Whenever a position is being fully removed via the burnPosition function, the full liquidity is being burned. That is exposed to a griefing vector, which is trivially possible if another user frontruns the call and uses 1 wei of the liquidity for option minting.
Recommendations	A usual fix would be to downcast the provided liquidity parameter in case it is larger than totalLiquidity - liquidityUsed. **However, we recommend acknowledging this issue in an effort to not introduce further changes.**
Comments / Resolution	Acknowledge.

Issue_74	Malicious hooks can be leveraged to prevent settlement
Severity	**Low**
Description	Any user can create a position with a malicious hook which reverts for settlement or reverts for other callpaths. This would essentially break most functionality for this position.
Recommendations	Consider only allowing for whitelisted hooks.
Comments / Resolution	Resolved, while it is still possible to create a position with any hook, these cannot be actively used for options because the Option contract indeed properly validates the hook during the mintOption function.