BAIL security

Gamma
UniswapV4 Integrations

FINAL REPORT

February ‘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 Gamma V4
Website	app.gamma.xyz
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/GammaStrategies/UniswapV4-Hypervisor/tree/9d2355574f6aa581dc23da6a0c9801acf3ef1d12/src
Resolution 1	https://github.com/GammaStrategies/UniswapV4-Hypervisor/tree/da9881dcbb9b2413bebb0e46a1f2a913c0830975/src
Resolution 2	https://github.com/GammaStrategies/UniswapV4-Hypervisor/tree/d4f38200473c19d8759d2b218d1fe85dc6b241d5 (unvalidated)

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	5	4		1	
Medium	8	7		1	
Low	7	2		5	
Informational	7	4		3	
Governance	1			1	
Total	28	17		11	

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

Disclaimer: During Resolution 1 at least one high severity bug was found plus the architecture was refactored to a large degree by outsourcing several functions from the MultiPositionManager to the PoolManagerUtils library. While all resolution changes have been verified and corresponding comments have been made, it is impossible to fully validate correctness due to the outsourcing which was made. In the current state, we cannot consider the architecture as safe. Bailsec has a very strict due diligence check on what can be validated with high confidence during the resolution round. This allows us to offer the highest possible security standards and outperform all of our competitors. In the scenario where no full confidence can be reached during a resolution round due to refactoring or too many changes, it is necessary to schedule a new audit round.

It is important that neither the commit da988 (resolution 1) nor the commit d4f38 (resolution 2) can be considered as validated by Bailsec.

3. Detection

PoolManagerUtils

The PoolManagerUtils library is used by the MultiPositionManager contract and is responsible for safe interactions with Uniswap V4's PoolManager contract. Specifically, it handles the following actions:

a) Addition of liquidity for desired positions via mintLiquidityForAmounts. This is triggered during a rebalance or a compound

b) Removal of liquidity proportional to the provided share amount and the total supply via the burnLiquidityForShare function. This is triggered during a rebalance or a standard liquidity pull action

c) Zero burn to update fees via the zeroBurnWithoutUnlock function. This is triggered before all interactions as well as during the ZERO_BURN operation.

d) Closing the pair via settling any debt or taking and credit based on the ephemeral balance. This function is called after any execution:

   i) WITHDRAW
   ii) ZERO_BURN
   iii) REBALANCE
   iv) COMPOUND
   v) PULL_LIQUIDITY
   vi) CLAIM_FEE

e) Converting liquidity in a range and a price to the corresponding tokenX/tokenY amount using the standard getAmountsForLiquidity function.

f) Fetching the tokenX/tokenY amount including fees based on the range, liquidity and current sqrtPrice. This is done within the getAmountsOf function.

Core Invariants:

INV 1: mintLiquidityForAmounts must only call modifyLiquidity if the sqrtPrice, range, amountX/Y setup results in non-zero liquidity

INV 2: amountX/amountY return value from modifyLiquidity during liquidity addition must never be below inMin

INV 3: burnLiquidityForShare must proportionally burn the whole existing liquidity based on the provided shares amount and totalSupply

INV 4: amountX/amountY return value from modify during liquidity removal must never be below outMin

INV 5: All positions must always have the same salt of bytes32(uint256(1))

INV 6: During burnLiquidityForShares and mintLiquidityForAmounts, the modifyLiquidity return value must never incorporate any pending fees

INV 7: zeroBurnWithoutUnlock must only call modifyLiquidity if there are unclaimed fees

INV 8: getAmountsOf must always include the accurate fee amount

Privileged Functions

- none

No issues found.

MultiPositionManager

The MultiPositionManager is a liquidity manager for UniswapV4 which is heavily inspired by the V3 Manager. It essentially allows users to deposit tokenX/tokenY into the contract for a corresponding amount of shares, while the tokenX/tokenY ratio must always match exactly the existing ratio in the system.

The mathematical formula behind the deposit calculation is as follows:

> depositXAmount = totalX * desiredShares / totalShares

> depositYAmount = totalY * desiredShares / totalShares

Whenever funds are sitting in the vault, governance can then add liquidity to multiple base positions as well as to two limit positions.

Once a withdrawal is executed, the user will simply receive the proportional amount of tokens based on the provided share amount and the totalSupply. This amount will be proportionally withdrawn from all existing liquidity positions and the idle balance.

Shares are denominated as ERC20 tokens and can then be used for potentially other purposes but will always reflect the underlying tokenX/tokenY amounts based on the current sqrtPrice of the pools, the ranges and corresponding liquidity.

The management of the vault is solely handled by governance via the following functions:

- rebalance
- compound
- pullLiquidity

Furthermore, the contract owner can not only determine multiple base ranges but also set a limitWidth which will add limitPositions starting from [baseTick + tickSpacing; baseTick + tickSpacing + limitWidth] for tokenX and [baseTick - limitWidth; baseTick] for tokenY, a thorough appendix can be found below.

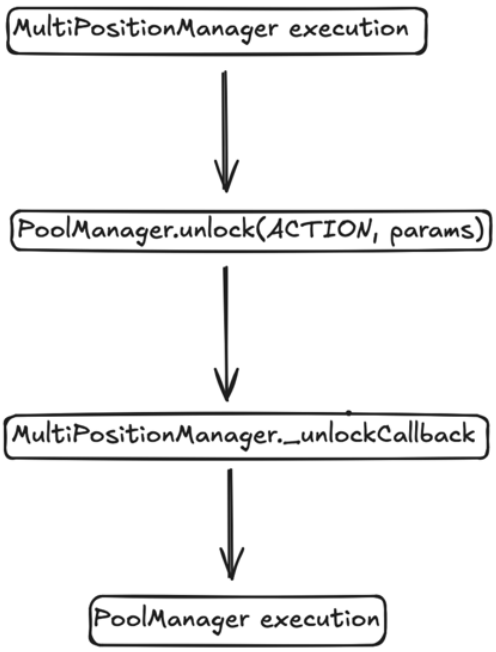
The contract applies a performance fee which is 20% by default but can be set up to 100% and is taken from the accrued swap fees.

Appendix: Limit Position Determination

The contract exposes the _setLimitRanges function which determines how the right and left-sided limit positions are determined. This is done as follows:

- Round down the currentTick to be a multiple of tickSpacing
- Determine left-sided range as [baseTick - limitWidth; baseTick]. Due to the rounding down it ensures that baseTick <= currentTick which is a valid state for left-sided liquidity
- Determine right-sided range as [baseTick + tickSpacing; baseTick + tickSpacing + limitWidth]. Due to the addition of tickSpacing it ensures that even with maximum rounding, lowerTick > currentTick which is a valid scenario for right-sided liquidity.

The contract interacts with UniswapV4 via the unlock function which then re-enters into the MultiPositionManager's safeCallback function to execute pair actions. This process can be illustrated as follows:



Appendix: UniswapV4 compatibility

In comparison to UniswapV3, the UniswapV4 protocol introduces some novel new features which must be incorporated into the MultiPositionManager to make the integration compatible, below we will shortly explain all important features:

Ephemeral balance: In UniswapV3, tokens are transferred in whenever liquidity is minted and tokens are transferred out whenever liquidity is burned. In the new V4 version, the token transfer is limited to the end of the execution. This means if liquidity is burned, tokens are not immediately transferred out but are instead accrued as positive delta into the transient storage via the _accountDelta function. Vice-versa, whenever liquidity is minted, this will be written into the transient storage as negative delta.

Unlock mechanism: In UniswapV3, it was possible to simply call mint/burn/swap/.. directly without any prior execution. However, in UniswapV4, it is necessary to call the unlock function on the PoolManager first which then allows to execute all desired operations. At the end of the unlock execution, it is then checked that the ephemeral balance in the transient storage for all tokens is zero, which means it is not allowed to have any leftover tokens. That means anyone who integrates with UniswapV4 must always settle all delta balances either via take or settle.

Fee claiming: In UniswapV3, fees were just added to the unclaimed tokenX/tokenY amount of a position. In UniswapV4, these are added as positive delta to the ephemeral balance. There are then two possibilities to claim these fees. Either simply via the take function or via the mint function which mints an ERC6090 token corresponding to the currency and decreases the ephemeral balance. This token can then be later redeemed via the burn function which transfers the underlying balance to a desired recipient address.

Position Salt: In UniswapV3, it was only possible to create one position for each range setting. In the V4 version, it is now possible to create multiple positions with the same range which is allowed due to the salt introduction into the key:

> calculatePositionKey(owner, tickLower, tickUpper, salt)

Donation: UniswapV4 implements a donate function which allows to donate funds which then increases feeGrowthGlobal. In UniswapV3 this was only possible by swapping back and forth (which essentially had the same effect as long as done without position crossing)

Hooks: UniswapV4 introduces the concept of hooks which allow for custom execution during certain interactions.

Summarized, the following hooks are possible in UniswapV4 pools:

- beforeInitialize: simple call; no return data
- afterInitialize: simple call; no return data
- beforeAddLiquidity: simple call; no return data
- afterAddLiquidity: returns hookDelta which allows for manipulating the ephemeral balance change; depending on DELTA_FLAG
- beforeRemoveLiquidity: simple call; no return data
- afterRemoveLiquidity: returns hookDelta which allows for manipulating the ephemeral balance change; depending on DELTA_FLAG
- beforeSwap: allows for fee override (dynamic fee); returns Delta which is used in afterSwap parameter; depending on DELTA_FLAG
- afterSwap: returns hookDelta which allows for manipulating the ephemeral balance change; depending on DELTA_FLAG
- beforeDonate: simple call; no return data
- afterDonate: simple call; no return data

However, this protocol is only compatible with pools that do not introduce hooks.

Core Invariants:

INV 1: initRatio must be always set to the corresponding tokenX/tokenY decimals

INV 2: The deposit function must always be called by a contract which updates the pool state and validates the from address

INV 3: _mintLimitPositions always consumes the full leftover tokenX/tokenY amount

INV 4: _getTotalAmounts must include unclaimed fees deducted by protocol fee

INV 5: Within _setLimitRanges, baseTick must always be rounded down in case it is not a multiple of tickSpacing

INV 6: _getCurrencyDeltas must always add ephemeral balances to idle balances

INV 7: _getTotalAmounts must include amounts from all existing base and limit positions as well as idle funds

INV 8: At the end of any pair operation, the ephemeral balance must always be zero

INV 9: Before any pair interaction, _zeroBurnAllWithoutUnlock must be called

INV 10: REBALANCE must fully burn the whole liquidity for every base and limit position

INV 11: PULL_LIQUIDITY must fully burn the whole liquidity for one position

INV 12: _burnLiquidities must loop over all base and limit positions

INV 13: All imported UniV4 libraries are expected to work flawless

INV 14: Within _getTotalAmounts, each position must be unique

INV 15: Within _burnLiquidities, each position must be unique

Privileged Functions

- transferOwnership
- renounceOwnership
- setWhitelist
- setFee
- setFeeRecipient
- pause
- unpause
- setInitRatio
- rebalance
- compound
- pullLiquidity
- claimFee

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. For example, it is possible to steal all funds via clever rebalances.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_02	msg.value is not deducted from _getTotalAmounts
Severity	**High**
Description	In the scenario where token0 is a native token and a deposit is executed, the provided msg.value will already be part of the contract balance. In the previous iteration this was correctly deducted from the _getTotalAmounts return values. However, in the current iteration this is not deducted, which will inflate the total AUM value and thus results in a loss for depositors. It will also have potential side-effects for the very first depositor.
Recommendations	Consider simply decreasing the existing balance of token0 by the msg.value in this scenario.
Comments / Resolution	Resolved, while it is confirmed that this bug was fixed by the resolution 2 commit, we need to emphasize that the resolution 2 commit cannot be considered as safe.

Issue_03	Funds can be stolen in double position edge-case
Severity	**High**
Description	The _mintLiquidities function allows for the addition of multiple basePositions and two limitPositions. There is the rare possibility that these positions share the exact same poolKey and lowerTick/upperTick. This can happen in two scenarios: a) Governance accidentally added two basePositions with the same credentials b) A limitPosition becomes the same range as a basePosition. This can happen in the scenario where a small tick deviation is allowed and the current tick changes: a) currentTick = 0, baseLower = -10, baseUpper = 3 (which is a legit setup for a base pos) with deviation allowance = 3 b) transaction is executed but attacker swapped currentPrice to 3 c) basePos is added with [-10; 3] d) limit pos has a width of 13 e) limitLower = 3 - 13 = -10 f) limitUpper = currentTick = 3 Thus, the basePosition has the exact same credentials as the limitPosition. This is a legitimate state which can happen even without a governance mistake. In itself, this is not an issue, however, it becomes an issue due to the fact how the withdraw function works. A loop is first executed over all basePositions to withdraw the proportional amount of shares for

	the existing liquidity of each basePosition and the same is done for both limitPositions. The flaw now lies within the fact that two positions are equal and now within the consultation of the basePosition liquidity, it will also include the limitPosition liquidity, since both are exactly the same. Thus, it will withdraw more proportional liquidity than it should. **Illustrated:** Status Quo: 50% of shares are burned basePos[0] = 100e18 liq basePos[1] = 150e18 liq user should receive pro-rata of 250*0.5 = 125 in reality receives: iteration 1: liquidity = 250e18 50% of it -> 125e18 iteration 2: liquidity = 125e18 50% of it = 62.5e18 user receives 187.5e18 instead of 125e18 This essentially allows a user to steal funds from the contract.
Recommendations	Consider: a) Making sure there are never the same basePositions b) Implementing a safeguard which ensures that abs(tickLower-tickUpper) for basePositions is never equal to limitWidth (after

	limitWidth rounded by tickSpacing)
Comments / Resolution	Resolved, a _checkRanges function has been incorporated which ensures that during rebalance no duplicate positions are created.

Issue_04	Incorrect calculation within _getTotalAmounts will result in loss for depositor in double position edge-case
Severity	**High**
Description	In the above mentioned issue, we have already explained that it is possible to have positions which have the completely same credentials. This will also have unexpected side-effects to the _getTotalAmounts function as it will essentially execute the same loop but liquidity will return the aggregated liquidity for both positions, which essentially double counts it and results in a loss of shares for the user. This is specifically a problem if a rebalance is executed afterwards since the new position setup will most likely not have any double positions anymore.
Recommendations	Consider: a) Making sure there are never the same basePositions b) Implementing a safeguard which ensures that abs(tickLower-tickUpper) for basePositions is never equal to limitWidth (after limitWidth rounded by tickSpacing)
Comments / Resolution	Resolved, a _checkRanges function has been incorporated which ensures that during rebalance no duplicate positions are created.

Issue_05	Deposit amounts are rounded down
Severity	**High**
Description	Currently, the deposit function calculates the necessary tokenX/Y amounts as follows: > totalX * desiredShares / totalShares > totalY * desiredShares / totalShares which is written in solidity as follows: *deposit0 = FullMath.mulDiv(pool0, shares, total);* *deposit1 = FullMath.mulDiv(pool1, shares, total);* This can be abused to receive shares for free, specifically in the scenario where one token has low decimals: 100000e6 * 0.0000009e18 / 100000e18 Moreover, there is already a battle-tested deposit function which enforces the share ratio mandate which is live since several years: https://github.com/charmfinance/alpha-vaults-contracts/blob/main/contracts/AlphaVault.sol#L167 Furthermore, if init0/1 is zero and there are insufficient checks within the UniProxy contract, a user can provide uint256.max as a shares parameter which allows the user to receive shares of uint256.max without providing any tokens. A part of these shares can then be redeemed which then allows for further deposits, tricking the subsequent depositor to donate a part of his funds to the first depositor.
Recommendations	Consider following the linked battle-tested approach.
Comments / Resolution	Failed resolution, the getTotalAmounts function returns amount0 including msg.value in case token0 is the native token.

	Resolution 2: Resolved, while it is confirmed that this bug was fixed by the resolution 2 commit, we need to emphasize that the resolution 2 commit cannot be considered as safe.

Issue_06	Using pairs with hooks can result in loss of funds
Severity	**High**
Description	Within the corresponding appendix we have explained the concept of hooks and the potential side-effects. This can result in a loss of funds or a DoS, specifically for if a pair with delta hooks is used. There are several other potential unexpected state transitions in the scenario where a pair with hooks is used.
Recommendations	Consider never depositing funds into a pair with hooks.
Comments / Resolution	Acknowledged, Gamma has ensured that no pools with hooks will ever be used.

Issue_07	Skipping of limit positions is not possible
Severity	**Medium**

Description	The _setLimitRanges function exposes the following condition:
	
	if (limitWidth == 0) {
	delete limitPositions;
	return;
	}
	
	This condition intends to only add basePositions and skip the limitPosition addition. It will however never work because the _mintLiquidities function always calls _mintLimitPositions which forcefully attempts to add these positions. This will however only revert because it attempts to fetch a Position which is non-existing:
	
	if (currencyDelta1 > 0) {
	PoolManagerUtils.mintLiquidityForAmounts(
	poolManager,
	limitPositions[0],
	0,
	currencyDelta1,
	[uint256(0), uint256(0)]
	);
	}
	
	(uint160 sqrtPriceX96, , ,) = poolManager.getSlot0(position.poolKey.toId());
	uint128 liquidity = LiquidityAmounts.getLiquidityForAmounts(
	sqrtPriceX96,
	TickMath.getSqrtPriceAtTick(position.lowerTick),
	TickMath.getSqrtPriceAtTick(position.upperTick),
	amount0,
	amount1
	);
	
	Thus, it is never possible to avoid the addition of limitPositions. This

	will have several side-effects, including but not limited to: - Rebalancing is never possible without adding the full idle balance - Compound is never possible without adding the full idle balance This seems to be a serious limitation of the business logic and will likely heavily impact the strategy flexibility.
Recommendations	Consider simply avoiding the forceful addition of limit positions in case there is no limitPosition set. This could be trivially implemented by an early return within _mintLimitPositions.
Comments / Resolution	Resolved, it however is important to mention that the codebase must be adjusted to implement early returns to prevent accidentally wrong storage collisions.

Issue_08	Lack of validation for fee
Severity	**Medium**
Description	The setFee function allows for setting the fee . Currently, there is no validation which ensures that the setting will result in a reasonable value.
Recommendations	Consider adding a validation to that value, including a zero-check.
Comments / Resolution	Resolved.

Issue_9	Fee change will impact state in hindsight
Severity	**Medium**
Description	The setFee function allows for setting the fee. Currently, the contract is not updated before the fee change which means that any fee change will be applied in hindsight, impacting expected values.
Recommendations	Consider updating the state before the fee change.
Comments / Resolution	Resolved.

Issue_10	Lack of pausing enforcement
Severity	**Medium**
Description	The contract inherits the Pausable contract but does not expose modifiers for it. Thus it is not possible to prevent withdrawals in emergency situations.
Recommendations	Consider implementing the whenNotPaused modifier.
Comments / Resolution	Resolved.

Issue_11	PULL_LIQUIDITY action will never work
Severity	**Medium**
Description	The PULL_LIQUIDITY action allows the owner to completely remove one desired position and transfer funds to the vault. This action will simply never work because it expects a return value: *bytes memory result = poolManager.unlock(* *abi.encode(Action.PULL_LIQUIDITY, params)* *);* *(amount0, amount1) = abi.decode(result, (uint256, uint256));* However, within _executeActionWithoutUnlock, no return value is actually provided for this action, thus it will always revert by trying to decode an empty return value in two uints.
Recommendations	Consider simply removing the return value assumption and the decoding.
Comments / Resolution	Resolved.

Issue_12	Deposit can result in a loss of ETH
Severity	**Medium**
Description	The deposit function allows to provide a msg.value even if currency0 is not native ETH. There is currently no sanity check within deposit or _transferIn which would prevent this.
Recommendations	Consider implementing a sanity check to prevent providing a msg.value if currency0 is not ETH
Comments / Resolution	Resolved.

Issue_13	Lack of deadline enforcement during deposit
Severity	**Medium**
Description	The deposit function is usually guarded via the UniProxy contract which ensures a proper slippage implementation. However, there is no deadline check which can result in the deposit being executed while the slippage check is met but potential opportunity costs can occur if the transaction stucks in the mempool versus a direct execution where the user would potentially have a more favorable outcome.
Recommendations	Consider implementing a deadline check within the UniProxy. Keep in mind that this contract is not included in the audit scope and thus this issue will always be marked as acknowledged with a corresponding comment.
Comments / Resolution	Acknowledged.

Issue_14	Denial of service due to bad practice within the Sync function
Severity	**Medium**
Description	Currently, the _closePair function calls the close function on both tokens (first token0, then token1) which will call the sync function, transfer the token and then the settle function. However, if the token is the address(0), only the settle function will be called which will call the transient storage to find the address of the token, which will be address(0) most of the time. However, if within the same transaction, multiple actions are done (for example, by batching them on a multisig) the transient storage value might be set to the address of a previous token in case only the settle function was called. Within this specific edge case, the call will revert and can cause a temporary DOS.

Recommendations	An easy way to fix this issue is to simply close the token1 first, then token0. Reasoning is that only token0 could be address(0), and closing on any ERC20 (token1) will call the correct flow: sync, transfer then settle. Settle will effectively reset the transient storage value and the call will go as expected.
Comments / Resolution	Resolved.

Issue_15	Potential storage collisions in case of empty limitPositions
Severity	**Low**
Description	Whenever limitPositions are empty, they will be considered as 0,0,0. Within the codebase, there are multiple spots where the storage is accessed based on these limitPositions. While for now it seems that no collision is possible, this still introduces a significant risk to the codebase which can be easily avoided.
Recommendations	Consider simply returning early for all spots in the codebase where an attempt is made to fetch storage based on limitPositions which are empty.
Comments / Resolution	Resolved, while it is confirmed that this bug was fixed by the resolution 2 commit, we need to emphasize that the resolution 2 commit cannot be considered as safe. The currentTicks function does not implement the early return pattern but it is expected that the returned tick is zero.

Issue_16	Lack of fee incorporation in getPositions function
Severity	**Low**
Description	The getPositions function completely ignores any unclaimed fees which results in the tokenX/tokenY amount being smaller than expected. This will also result in an incorrect return value within getOutMinForShares.
Recommendations	Consider incorporating the fee (deducted by the protocol fee).
Comments / Resolution	Acknowledged.

Issue_17	Limit positions are always tied to the current price
Severity	**Low**
Description	The way how _setLimitRanges works is by setting the left and right sided limit position around the currentTick (with some rounding and safeguards). This will essentially not allow for placing limit orders at a higher starting price than the current price (for the curve setting).
Recommendations	Consider if more flexibility for this function is desired. The implementation of such flexibility will deviate the _setLimitRanges position function and most likely requires careful additional validation time to ensure no side-effects will be introduced from that. In our opinion, this issue can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_18	Lack of validation for init0/1
Severity	**Low**
Description	The setInit function allows for setting the init0/1 parameters . Currently, there is no validation which ensures that the setting will result in a reasonable value. If these values are set too low, it is possible that the share denomination becomes very high and at some point results in overflow.
Recommendations	Consider implementing a reasonable validation.
Comments / Resolution	Resolved, this function has been removed.

Issue_19	Incorrect view-only return value from getTotalAmounts if fee is changed
Severity	**Low**
Description	The getTotalAmounts function returns tokenX/tokenY for all positions including unclaimed fees, deducted by the protocol fee. If the protocol fee is ever changed before a zero burn is executed, this will result in a falsified return value b/w pre-fee change consultation of getTotalAmounts and post-fee change consultation.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_20	Liquidity can be pulled out at no cost
Severity	**Low**
Description	Similar to most other V3/4 vaults which do not execute direct deposits, a user can execute a deposit and a withdrawal in the same transaction which then pulls out liquidity from the pair, essentially griefing the functionality.
Recommendations	Consider acknowledging that issue and implementing additional safeguards within the UniProxy if that is ever abused.
Comments / Resolution	Acknowledged.

Issue_21	Edge-conditions within _roundUp can result in unexpected side-effects
Severity	**Low**
Description	The _roundUp function handles several different conditions: *if (tickLower < TickMath.MIN_TICK) {* *tickLower = TickMath.MIN_TICK;* *}* *if (tickUpper > TickMath.MAX_TICK) {* *tickUpper = TickMath.MAX_TICK;* *}* *if (tickLower >= tickUpper) {* *return (0, 0);* *}* While it is theoretically possible that these conditions are reached due to rounding and / or application of limitWidth, the contract should in the first place never be in such a state to be near the boundary ticks when adding liquidity. Thus to be completely safe we simply recommend to directly revert

	if one of these scenarios is triggered.
Recommendations	Consider reverting for these conditions.
Comments / Resolution	Acknowledged.

Issue_22	Several possible side-effects if deposit is not executed via UniProxy
Severity	**Informational**
Description	The deposit function is meant to be called by a whitelisted address only which is here the UniProxy contract. If the deposit function is ever callable via another contract, several possible side-effects can happen (including but not limited to): - Abuse of the “from” address - Deposit without zero burn which results in side-effects if fee is changed post-deposit - Potentially incorrect refund of ETH to msg.sender - Loss of funds due to lack of slippage check This issue is only rated as informational because it is expected that the UniProxy is always used as a guard.
Recommendations	Consider always calling the deposit function via the UniProxyV2 contract.
Comments / Resolution	Acknowledged.

Issue_23	Deposits are blocked until init0/1 settings
Severity	**Informational**
Description	Currently, it is not possible to execute the first deposit if init0/1 is not set. This can result in a delay of the launch if these parameters are forgotten to be set.
Recommendations	Consider setting these parameters upon deployment.
Comments / Resolution	Resolved.

Issue_24	Out of gas possibility due to unlimited basePosition size
Severity	**Informational**
Description	The _burnLiquidities function loops over basePositions in the scenario where the array becomes unreasonably large, this loop will consume excessive gas which can result in a function revert due to the “out of gas” issue.
Recommendations	Consider ensuring that a reasonable amount of basePositions is added.
Comments / Resolution	Acknowledged.

Issue_25	Redundant logic within _getCurrencyDeltas
Severity	**Informational**
Description	The _getCurrencyDeltas function handles the following scenario: *if (delta0 < 0 || delta1 < 0) revert CurrencyDeltaError();* This is never actually reached in the first place because the subtraction will revert due to an underflow because delta0/1 is a uint.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Resolved, against the recommendation.

Issue_26	Unused variable(s)
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. *uint256 constant PRECISION = 1e36;* *error InvalidPoolKey(PoolKey key);*
Recommendations	Consider removing all unused variables.
Comments / Resolution	Resolved.

Issue_27	Transfer-tax incompatibility
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using such tokens
Comments / Resolution	Acknowledged.

Issue_28	Redundant dups mapping
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers: *mapping (bytes32 => bool) dups;*
Recommendations	Consider removing this variable.
Comments / Resolution	Resolved.