BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Impermax
V3 Core

January 2024

Disclaimer:

1. Project Details

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

Project	Impermax V3
Website	impermax.finance
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/Impermax-Finance/impermax-v3-core/tree/d78898deeff1ac174c4b12328fcdd874fcccb896
Resolution 1	https://github.com/Impermax-Finance/impermax-v3-core/tree/4978c2ecdb59fadd91751fa848eb9336511f2f85/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	13	10		3
Medium	16	4		12
Low	13	2		11
Informational	1			1
Governance	3	1		2
Total	46	17		29

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

Due to the amount of high risk findings, we highly recommend another audit by an independent team!

3. Detection

Disclaimer: ImpermaxV3 is not meant to be used with transfer-tax tokens

V3Factory

The V3Factory module is responsible for the permissionless deployment of a lending pool. The following modules are part of each lending pool:

- TokenizedV3 Module: This module allows users to tokenize their Uniswap V3 liquidity positions.

- Collateral Module: This module allows users to deposit their TokenizedV3 position which then signals that it can be used as collateral.

- Borrowable Module: This module allows lenders to deposit tokenX/tokenY and borrowers to borrow tokenX/tokenY using their tokenId as collateral. For this module, there are two deployments as one deployment is tied to tokenX and the second to tokenX

The deployment process works as follows:

a) Deploy TokenizedV3 module for tokenX/tokenY
b) Deploy collateral module for corresponding TokenizedV3 module
c) Deploy borrowable0/1 for corresponding TokenizedV3 module
d) Initialize the lending pool once all modules have been deployed and linked

ImpermaxV3Factory

The ImpermaxV3Factory is responsible for deploying the collateral and borrowable module towards the corresponding ImpermaxUniswapV3Position contract. Once both modules are deployed, they can be initialized permissionless.

Privileged Functions

- _setPendingAction
- _acceptAdmin
- _setReservesPendingAdmin
- _acceptReservesAdmin
- _setReservesManager

Issue_01	Borrowable and Collateral module can be deployed tied to malicious UniswapV3TokenizedPosition
Severity	**High**
Description	The usual deployment process would be to first deploy the TokenizedV3 module and then the borrowable and collateral module. This furthermore allows for the deployment of malicious UniswapV3TokenizedPosition contracts with a custom TokenizedV3 implementation which can then be used to trick users into lending tokens into the borrowable module such that the malicious deployer can now borrow all funds because the malicious tokenizedV3 implementation shows unlimited collateral.
Recommendations	Consider only allowing to deploy the borrowable and collateral module for an nftLp address which was already deployed by the TokenizedUniswapV3Factory.
Comments / Resolution	Acknowledged, any pool which is not vetted by the Impermax team will not be displayed on the frontend.

Issue_02	Lack of _getTokens check within createCollateral; createBorrowable0/1
Severity	**Medium**
Description	Currently, the _getTokens check within all three mentioned functions has been removed. This has been an important sanity check which ensures that the nftlp address is in fact an already deployed address which exposes both token variables. With this new approach, it is now possible to deploy the collateral and borrowable module even before the tokenized module has been deployed. More importantly, this can even be done for legitimate tokenized modules since the deployment address is deterministic. 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 re-adding the _getTokens check.
Comments / Resolution	Acknowledged.

BDeployer

The BDeployer contract is tied to the ImpermaxV3Factory and responsible for the deployment of ImpermaxV3Borrowable contracts via low-level create2 calls.

Privileged Functions

- none

No issues found.

CDeployer

The CDeployer contract is tied to the ImpermaxV3Factory and responsible for the deployment of ImpermaxV3Collateral contracts via low-level create2 calls.

Privileged Functions

- none

TokenizedV3Factory

TokenizedUniswapV3Factory

The TokenizedUniswapV3Factory allows the permissionless deployment of TokenizedUniswapV3Position contracts which can then be used to tokenize a V3 position.

Privileged Functions

- _setPendingAdmin
- _acceptAdmin
- _setAcModule

Issue_03	Legitimate deployment can use newly created UniswapV3Pool with manipulated TWAP price
Severity	**High**
Description	A malicious user can deploy a new UniswapV3Pool for a famous token pair which has no pool yet and keep the price in an artificial state for 1800 seconds. This works because there is no incentive to MEV a newly deployed pool with no/low liquidity. After that, the user can deploy a new TokenizedUniswapV3Position contract and the borrow + collateral module to set up the pool. The user can then create a single sided LP position only with tokenX to benefit from the inflated/malicious TWAP price, wait until users lend tokenY into the borrow module and drain it.
Recommendations	This issue is inherent to the permissionless deployment process and does not rely on a malicious TokenizedUniswapV3Position implementation. Therefore, we recommend implementing a manual due diligence process for frontend listings which includes inspecting the historic TWAP price of a newly deployed pool.
Comments / Resolution	Resolution 1: This has been refactored to the usage of the Chainlink oracle. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved.

TokenizedUniswapV3Deployer

The TokenizedUniswapV3Deployer contract is tied to the TokenizedUniswapV3Factory contract and handles the low-level deployment of TokenizedUniswapV3Position contracts via create2.

Appendix: create2

- create2 is an EVM opcode that allows deploying a contract to an address determined by:
    1. The contract's **bytecode**,
    2. A **salt** (some arbitrary data),
    3. The **deploying address**.

Thus, the resulting contract address is **predictable** ahead of time, because it's basically keccak256(0xff, deployer, salt, keccak256(bytecode))[12..].

Privileged Functions

- none

No issues found.

Borrowable Module

ImpermaxERC20

The ImpermaxERC20 contract is a simple ERC20 contract which is part of the borrowable module and is minted to users upon depositing the underlying token. It basically serves as receipt for that token with the goal to accumulate borrow interest rewards and can be redeemed by users at any time for the underlying token (as long as there are sufficient non-borrowed tokens)

Privileged Functions

- none

Issue_04	Same nonce usage for borrow permit and standard permit
Severity	**Medium**
Description	Currently, the contract uses the same nonce for the borrow permit and approval permit validation: *bytes32 digest = keccak256(* *abi.encodePacked(* *'\x19\x01',* *DOMAIN_SEPARATOR,* *keccak256(abi.encode(typehash, owner, spender, value, nonces[owner]++, deadline))* *)* *);* That means, if a user signs a message which allows another user to increase the borrow allowance but at the same time the same user signs a message which is used to increase the standard ERC20 allowance, the second of both calls will now revert because it erroneously uses the same nonce.
Recommendations	Consider if it makes sense to implement different nonces for both purposes.
Comments / Resolution	Acknowledged.

PoolToken

The PoolToken contract allows users to deposit the underlying token in exchange for the ImpermaxERC20 token with the goal to accrue interest from borrowers. Users can redeem their ImpermaxERC20 tokens at any time, as long as there are enough free, unborrowed funds.

Appendix: Exchange Rate

The contract exposes an exchangeRate function which calculates the value of ImpermaxERC20 tokens similar to an ERC4626 vault:

> totalBalance * 1e18 / totalSupply

This formula is however overridden within the ImpermaxV3Borrowable contract to properly account for any borrowed funds (as during borrow it transfers out funds which would then decrease totalBalance). The updated formula is as follows:

> (totalBalance + totalBorrows) * 1e18 / totalSupply

Core Invariants:

INV 1: The borrow state must be fully updated at the beginning of mint and redeem

INV 2: exchangeRate and sync must be overridden

INV 3: The contract must only be used via a peripheral contract

INV 4: The amount of received ERC20 tokens must be non-zero

INV 5: The amount of received underlying tokens must be non-zero

Privileged Functions

- setFactory

Issue_05	Exchange rate calculation is vulnerable to first-depositor attack
Severity	**High**

Description

The mint function uses the standard ERC4626 formula to calculate the received amount of tokens. This formula is inherently vulnerable to the share inflation attack where a user can transfer tokens to the contract to then increase the divisor of the calculation:

amount * 1e18 / ((balance+totalBorrows)*1e18/totalSupply)

This function has two safeguards in place to mitigate this attack:

a) Mints 1000 shares to dead address which decreases the efficiency of this attack
b) Checks for zero-received tokens

These checks are good, however, they do not prevent this attack fully. A user can still execute this attack which results in rounding down tokens for subsequent depositors:

a) deposit 9009
    i) amount *1e18/1e18
    i) 9000
    ii) user receives 8000 token
    iii) 1000 token are burned

b) send 1000e18 token and sync
    i) totalBalance = 1000e18 + 9000
    ii) totalShares = 9000

c) Victim deposits 1.5e18
    i) amount * 1e18 / (balance*1e18/totalSupply)
    ii) 1.5e18 * 1e18 / ((1000e18+9000) * 1e18 / 9000)
    iii) receives 13.5 -> round down to 13
    iv) victim has lost tokens

	It is clear that it will take some time until attacker gains his funds back (since he initially donated some funds to dead addr) Victims will always lose some tokens due to rounding which then goes to the attacker.
Recommendations	Consider incorporating a simple slippage check.
Comments / Resolution	Acknowledged. The team will make the first deposit

Issue_06	Lack of contract address check in _safeTransfer
Severity	**Low**
Description	The _safeTransfer function lacks a valid contract address check. If the token is no contract, the call will not revert. This is against best-practices and can result in unexpected edge-cases. For example, users can borrow and increase the _totalBorrows variable without even having any tokens in the contract to borrow from (if underlyingToken is not yet deployed).
Recommendations	Consider following OpenZeppelin's safeERC20 usage.
Comments / Resolution	Resolved.

BStorage

The BStorage contract exposes the storage layout for the borrowable module.

Privileged Functions

- none

BAllowance

The BAllowance contract handles the borrow allowance.
Collateral owners can not only borrow on behalf of their own but also delegate allowance towards another address which then allows the other address to borrow on behalf of the collateral owner. It is important to mention that this is not tied to a specific tokenId but rather to the collateral owner address.

This can be done either via borrowApprove or via borrowPermit (using a valid signature).

Privileged Functions

- none

Issue_07	Allowance decrease can be frontrun
Severity	**Medium**
Description	The borrowApprove function simply resets the allowance to the new value: borrowAllowance[owner][spender] = value; If now the spender has a previous allowance of 100e18 and the owner wants to decrease this to 10e18, the spender can frontrun this call and consume the full 100e18 allowance, while getting an additional allowance of 10e18 granted.
Recommendations	Consider incorporating safeIncrease and safeDecrease functions.
Comments / Resolution	Acknowledged.

BInterestRateModel

The BInterestRateModel handles the interest accrual based on the time passed since the last update and the current borrowRate.

An interestFactor is built using the following formula:

interestFactor = borrowRate * timeElapsed

which is then applied on the total borrowed amount as follows:

interestAccumulated = interestFactor * totalBorrows

The accrued interest is reflected in the increased borrowIndex and totalBorrows variable:

totalBorrows + (interestFactor * totalBorrows / 1e18) borrowIndex + (interestFactor * borrowIndex / 1e18)

While the totalBorrows variable directly influences the exchangeRate for the PoolToken, the borrowIndex is used to determine the outstanding borrowed balance from a user including the interest since the beginning.

If for example a user has borrowed 100e18 tokens while the borrowIndex was 1e18 and the interest accrual increased the borrowIndex to 1.1e18, the user now owes the protocol 110e18 tokens.

The borrowRate is updated whenever _calculateBorrowRate is called which follows the following algorithm:

Adjust Kink Borrow Rate Over Time

- The contract tracks a baseline rate called kinkBorrowRate.
- If the last update was at _rateUpdateTimestamp, the code calculates timeElapsed = block.timestamp - _rateUpdateTimestamp.
- Based on whether the **current** borrowRate is **below** or **above** kinkBorrowRate, it applies an "adjust factor" that nudges the kink rate up or down, scaling by adjustSpeed and timeElapsed.

The updated _kinkBorrowRate is clamped within [KINK_BORROW_RATE_MIN, KINK_BORROW_RATE_MAX].

This approach gradually pulls the kink rate closer to the actual borrowRate if there's a gap, reflecting changing market conditions.

Calculate Utilization

If _actualBalance == 0, utilization = 0.

Otherwise, _utilizationRate = totalBorrows * 1e18 / _actualBalance.

Determine the new borrowRate based on the kink concept:

- **If utilization <= kinkUtilizationRate** (e.g., 75%), scale linearly:
    - _borrowRate=_kinkBorrowRate×utilizationkinkUtilizationRate /_borrowRate

- **If utilization > kinkUtilizationRate**, the rate grows more steeply
    - _borrowRate = ((KINK_MULTIPLIER - 1) * overUtilization + 1e18) * _kinkBorrowRate / 1e18
    - That effectively means any portion above the kink is multiplied to create a higher interest rate, encouraging borrowers to repay if the pool is highly utilized.

Core Invariants:

INV 1: totalBorrows and borrowIndex must always be increased by the same interestFactor

INV 2: totalBorrows and borrowIndex must always be set at the end of the accrueInterest function

Privileged Functions

- none

Issue_08	Protocol will be shut down if interest rate results in totalBorrows becoming larger than uint112.max
Severity	**Medium**
Description	Currently, the accrueInterest function safe wraps the totalBorrows value into a uint112: totalBorrows = safe112(_totalBorrows); This would be theoretically safe since the architecture is not meant to be used with tokens that have a supply larger than uint112.max. However, since the totalBorrows variable can in fact become > uint112.max even with tokens that have the fixed upper supply, it is possible that this safe wrap reverts which has the effect that the protocol is essentially being shut down.
Recommendations	Consider either removing the safe wrap and downscaling the totalBorrows value in that scenario or make sure only tokens are added where this can never happen (tokens which have a supply which is significantly lower than uint112).
Comments / Resolution	Acknowledged. The team will ensure that only tokens will be added where this cannot happen.

BSetter

The BSetter contract handles the setting of important protocol variables such as interest rate configuration, debt ceiling or the reserve rate

Privileged Functions

- _setReserveFactor
- _setKinkUtilizationRate
- _setAdjustSpeed
- _setDebtCeiling
- _setBorrowTracker

Issue_09	Governance: Setting of malicious borrowTracker can DoS contract
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 a malicious borrowTracker which does not match with the function selector and thus results in a revert of the _updateBorrow function.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Resolved. The borrowTracker logic has been removed

Issue_10	Change of interest rate parameters will be applied in hindsight
Severity	**Medium**
Description	The _setKinkUtilizationRate and _setAdjustSpeed functions change parameters for the interest rate calculation without updating the same before. This will result in these parameters influencing the interest calculation in hindsight instead of becoming only effective once adjusted.
Recommendations	Consider updating the protocol state before these parameters are changed.
Comments / Resolution	Acknowledged. The team will ensure that the contract state is updated before any parameter is changed.

Issue_11	Change of reserveFactor will mint more/less tokens to the reserve upon next state update
Severity	**Medium**
Description	The _setReserveFactor function sets the reserveFactor which is used within _mintReserves to calculate the reserve value from delta [ER; ERLast] which is minted to the reservesManager: `uint _exchangeRateNew = _exchangeRate.sub( _exchangeRate.sub(_exchangeRateLast).mul(reserveFactor).div(1e18) );` `uint liquidity = _totalSupply.mul(_exchangeRate).div(_exchangeRateNew).sub(_totalSupply);` `if (liquidity > 0) {` `address reservesManager = IFactory(factory).reservesManager();` `_mint(reservesManager, liquidity);` `}` Such a change will inherently impact the amount which is minted to the reservesManager not only onwards from the current block.timestamp but also in hindsight for the period b/w [last update; block.timestamp].
Recommendations	Consider invoking the exchangeRate function beforehand.
Comments / Resolution	Acknowledged. The team will ensure that the contract state is updated before the reserveFactor is changed.

Issue_12	Change of borrowTracker will potentially violate borrowTracker integrity
Severity	**Medium**
Description	The borrowTracker variable is used as an external contract to include logic for future reward distributions based on the positions. First of all, we need to acknowledge that this implementation is currently unavailable to review. However, it is most likely that the change of this variable will have an adverse impact on the reward calculation within the borrowTracker.
Recommendations	Consider being careful when changing this variable as well as keeping it in mind.
Comments / Resolution	Resolved, this functionality has been removed.

ImpermaxV3Borrowable

The ImpermaxV3Borrowable contract facilitates the borrowing and repaying of the underlying token for collateral providers. Users can borrow up to their variable LTV (which is based on safetyMargin and the composition of the collateral / debt position).

As soon as a position is considered as liquidatable but not underwater, anyone can liquidate this position by repaying the full borrowed amount to then gain an incentive on top of it. Collateral providers that get liquidated will always experience a 4% loss on their collateral which consists out of the liquidationIncentive and the protocolFee.

Appendix: Borrow Flow

The borrow function allows a user to borrow on behalf of an ImpermaxV3Collateral tokenId. This can be done if the user is the owner or if the user has an allowance for that tokenId.

The function first optimistically transfers out the token and then updates the borrow state of the tokenId and the overall borrowed amount. It subsequently considers the ImpermaxV3Collateral contract if the position is still healthy based on the updated borrow state and the collateral value.

The updated borrow state includes the debt on the pair's corresponding underlying token plus the updated debt based on the current underlying token where the loan is taken from.

Appendix: Repay Flow

Repayments can be done using the borrow function with or without borrowing funds. In essence it just requires an excess amount of tokens in the contract while the borrow function is invoked, combined with a lower borrowAmount parameter than the excess amount in the contract:

> (balanceOf + borrowAmount) - totalBalance

If only a repayment is done:

> (balanceOf + 0) - totalBalance

Edge-case:

If a repayment is done but at the same time funds are borrowed:

> (balanceOf + borrowAmount) - totalBalance

> A user has repaid 10e18 tokens and borrowed 5e18; balance before repayment = 100e18

> transfer 10e18 token in

> transfer 5e18 tokens out

> (105e18 + 5e18) - 100e18

> repaymentAmount = 10e18

> borrowAmount = 5e18

The _updateBorrow function now handles this edge-case internally.

Appendix: Bad debt restructuring

A position is considered underwater (in bad debt) as soon as:

value(collateral) < (value(debt) * penalty)

In that scenario, a liquidation is impossible and the bad debt needs to be removed from a system which is done via virtual repayment. All depositors will bear the loss in the form of a decreased exchangeRate.

If that happens, the position is restructured in such a way that it needs to be liquidated immediately in the same block. Otherwise, the debt will grow again due to the interest rate application which marks the position again as underwater and prevents liquidation.

The formula to calculate the value of the repayAmount is as follows:

> reduceToRatio = collateralValue * 1e18 / (collateralNeeded * liquidationPenalty)

> repayAmount = borrowBalance - ((borrowBalance * reduceToRatio) / 1e18)

The goal is that the position is afterwards fully liquidated:

> repayToCollateralRatio * liquidationPenalty / 1e18 = 1e18

Appendix: Reserve Factor

The contract works with an exchangeRate which is calculated as follows:

> (totalBalance + totalBorrows) * 1e18 / totalSupply

under normal circumstances, the exchangeRate steadily increases due to the increase of the totalBorrows variable from the interest accumulation.

Consider the following example to illustrate the increase:

> totalBalance = 100e18
> totalBorrows = 100e18
> totalSupply = 200e18
> exchangeRate = 1e18

After some time, interest is due and collected within the accrue function, which increases totalBorrows:

> totalBalance = 100e18
> totalBorrows = 110e18
> totalSupply = 200e18
> exchangeRate = 1.05e18
As one can see, the exchangeRate increased.

This is where the reserveFactor comes into play, as a part of the accumulated interest goes towards the protocol, depending on the reserveFactor value.

Whenever mint/redeem is called, this will call the exchangeRate function which then takes the accrued interest since the last mint/redeem into account and calculates the new exchangeRate:

> exchangeRate = (totalBalance + totalBorrows) * 1e18 / totalSupply

However, if the reserveFactor is non-zero, the exchangeRate update will be adjusted to reflect the amount which is minted to the reserve:

> newExchangeRate = ER - ((ER - ERLast) * reserveFactor / 1e18)

And the corresponding “to be minted” liquidity is calculated as follows:

> (totalSupply * ER / newExchangeRate) - totalSupply

Core Invariants

INV 1: borrowSnapshot.interestIndex must always be updated if a repay/borrow has happened

INV 2: If a user has fully repaid the borrowed amount, borrowSnapshot.interestIndex must be set to zero

INV 3: _totalBorrows must be always increased whenever interest is accrued

INV 4: _borrowIndex must be always increased whenever interest is accrued

INV 5: The contract is, similarly to UniswapV3, only compatible with tokens of a totalSupply from up to uint112.max

INV 6: A user can only borrow if he is the owner of the tokenId or has approval

INV 7: The TokenizedUniswapV3Position must be deposited into the ImpermaxV3Collateral contract in an effort to allow borrows

INV 8: Debt can only be repaid in the token it has been taken

INV 9: The tokenizedV3 position must be deposited in the collateral module in an effort to borrow on behalf of it

INV 10: exchangeRateLast must only be updated if reserves were minted

Privileged Functions

- none

Issue_13	Cross-contract reentrancy attack during borrow attack allows for draining the contract
Severity	**High**
Description	Users can borrow as long as they are the owner of the tokenId from the ImpermaxV3Collateral contract or as long as they have sufficient allowance for a tokenId. This can be done via the borrow function which optimistically transfers tokens out and then subsequently checks the collateralization of the tokenId. The following callback function allows for a cross-contract reentrancy attack where a malicious user can borrow funds while at the same time being able to redeem the TokenizedV3 tokenId fully: *if (data.length > 0) IImpermaxCallee(receiver).impermaxV3Borrow(msg.sender, tokenId, borrowAmount, data);* Process: a) Alice creates a tokenizedV3 position and deposits this position in the ImpermaxV3Collateral contract which now enables her to borrow up to the corresponding LTV b) Alice calls borrow which optimistically transfers out funds and reenters on the callback into the ImpermaxV3Collateral contract and calls the redeem function. c) Redeem passes since the borrowBalance at this point is not yet

	updated:
	
	redeemTokenId = tokenId;
	_burn(tokenId);
	INFTLP(underlying).safeTransferFrom(address(this), to, redeemTokenId);
	if (data.length > 0) IImpermaxCallee(to).impermaxV3Redeem(msg.sender, tokenId, redeemTokenId, data);
	
	// finally check that the position is not left underwater
	require(IBorrowable(borrowable0).borrowBalance(tokenId) == 0, "ImpermaxV3Collateral: INSUFFICIENT_LIQUIDITY");
	require(IBorrowable(borrowable1).borrowBalance(tokenId) == 0, "ImpermaxV3Collateral: INSUFFICIENT_LIQUIDITY");
	
	d) The borrow function now checks if the tokenId is sufficiently collateralized, which it in fact is. There is no such check which ensures that the tokenId is still owned by the ImpermaxV3Collateral contract.
	
	e) The exploiter now ends up with borrowed funds and an TokenizedV3 tokenId which can be fully redeemed for the underlying collateral.

Recommendations	There are multiple different approaches to fix this issue. The most trivial one is to simply execute an ownership check within the canBorrow function.
Comments / Resolution	Resolved.

Issue_14	Missing liquidation incentive in case TWAP deviates from SPOT
Severity	**High**
Description	Liquidations allow the liquidator to pay back the debt from a user and receive the equivalent value of the collateral plus the liquidationIncentive, which is by default 2%. This means for example if a user has a debt of 60 tokenY and a collateral of 100 tokenX and the TWAP price is 1, the user can repay 60 tokenY and receive 61.2 tokenX. Now consider a scenario where the SPOT price deviates from the TWAP price and is 0.9 instead of 1 a) The liquidator pays 60 tokenY which corresponds to 66.6 tokenX b) The liquidator receives 61.2 tokenX which corresponds to 55.08 tokenY c) The liquidator essentially made a loss on the liquidation and thus no liquidations will happen in that scenario. If the price is then further in a downside spirale, that means the SPOT price will always be smaller than the TWAP price and liquidations will be prevented for a prolonged time.
Recommendations	Consider using a strong oracle.
Comments / Resolution	Resolution 1: This has been refactored to the usage of the Chainlink oracle. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved

Issue_15	Downcasting of repayAmount during liquidate can be used to trick liquidators into losing funds
Severity	**High**
Description	The liquidate function allows a liquidator to repay a specific "repayAmount" and then receive the corresponding share of the repayAmount * liquidationIncentive in form of a tokenId from the collateral. If a liquidator for example repays 100e18 tokenX, it is expected that the user receives the equivalent of 10e18 * 1.02 tokenX in form of the collateral. There are two specific issues for this function: a) If a user transfers more in than the repayAmount parameter, the excess amount will be donated to the contract during the update modifier. The root-cause of that is the following loose check: *require(balance.sub(totalBalance) >= repayAmount, "ImpermaxV3Borrowable: INSUFFICIENT_ACTUAL_REPAY");* b) If a user provides a larger repayAmount than the actual debt of the position, the same issue as within "Excessive repayAmount will not be refunded to repayer" occurs. This is not only an issue per-say but can also be dramatically abused by the position owner to trick the liquidator into donating a large amount of tokens towards the contract which then flows into the increase of the exchangeRate. Trivially speaking, a user can just frontrun any liquidation call, repaying a part or the whole debt (while also withdrawing a part of the collateral), which then results in downcasting repayAmount because borrowBalance became significantly lower:

repayAmount = Math.min(repayAmount, borrowBalance(tokenId));

The downcast on the repayAmount does however not refund the previously provided funds back to the liquidator. In fact, these funds will simply be donated to the contract, which means the liquidator can lose up to all provided funds if the collateral owner decides to repay near the whole debt such that borrowBalance(tokenId) becomes near zero.

The collateral owner must execute the following steps before the liquidation call:

a) Repay almost all of the debt
b) Redeem the collateral of the tokenId to bring the position in such a state that it is not considered as liquidatable but on the brink of liquidation (see issue: Lack of safety threshold can result in immediate liquidation post-borrow)

c) Hope that the liquidators transaction is executed only in the subsequent block such that the interest accrual b/w the current block and the next block will result in the position becoming liquidatable (or the price change)

d) The liquidator now repays a large amount which is downcasted to almost zero, while receiving only dust in exchange.

The contract now compounds the repayment amount into the exchangeRate which can be abused by the collateral owner if he owns a large lending position.

This can also be executed with the incorporation of a block stuffing attack which means it is guaranteed that the liquidator's transaction will only be executed in the subsequent block.

	Furthermore, if a user is already liquidatable but not yet in bad debt, the user can just repay a part of the debt such that the position is still considered as liquidatable, which will have the same (but less dramatic) effect.
Recommendations	Consider refunding any excess repayment amount or executing a strict check which ensures that balanceOf - totalBalance does not exceed repayAmount by a large margin (one could implement a small deviation threshold).
Comments / Resolution	Acknowledged, the team added the following comment: Liquidations are handled by external contracts. The external contract should check the currentBorrowBalance of the position before calling the liquidate() function. This is in line with the philosophy that "the core contracts should only implement the minimal logic that allows them to do their job", additional checks over that must be handled by periphery contracts.

Issue_16	Permanently locked funds if reservesManager is address(0) while reserveFactor is non-zero
Severity	**High**

Description	Within the _mintReserves function, a part of the accumulated interest b/w [exchangeRateLast; _exchangeRate] is being minted as liquidity to the reserveManager address. A full overview of this mechanism can be found within (Appendix: Reserve Factor).
	
	If the reserveManager is address(0), it will mint the ImpermaxV3Borrowable token to address(0), as there is no address(0) sanity check within the _mint function:
	
	function _mint(address to, uint value) internal {
	totalSupply = totalSupply.add(value);
	balanceOf[to] = balanceOf[to].add(value);
	emit Transfer(address(0), to, value);
	}
	
	These funds are permanently unretrievable and will just decrease the exchangeRate (resulting in a loss for depositors) for no reason.

Recommendations	Consider ensuring that reserveManager can never be address(0).
Comments / Resolution	Resolved.

Issue_17	TWAP <-> SPOT price deviation allows for borrowing more than the collateral is worth if safetyMargin low
Severity	**Medium**
Description	Currently, the safetyMargin is considered as the LTV of the protocol and is in its most strict state when a user borrows asymmetric (collateral in tokenX, borrows tokenY). At the same time, this is also a state where the contract is most vulnerable in the case where the SPOT price deviates from the TWAP price, as users could abuse that scenario to deposit the “higher valued” token as collateral and borrow the “cheaper valued” token. This is usually countered by the LTV but if the safetyMargin is insufficiently strict or the pair experiences large volatility (large drop/increase within a few blocks), this increases the possibility of such an attack.
Recommendations	Consider using a strong oracle.
Comments / Resolution	Resolution 1: This has been refactored to the usage of the Chainlink oracle. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved.

Issue_18	Excessive repayAmount will not be refunded to repayer
Severity	**Medium**
Description	As explained in the contract description, the borrow function is not only used to borrow funds but also to repay outstanding debt. This can be done by either transferring funds in before the function is being called or during the callback. If a user repays an amount which is above the debt of a position, this amount will not be refunded to the repayer. Instead it will just sync'd at the end of the function. This is due to the way how an excessive repayment is handled within the _updateBorrow function: *accountBorrows = accountBorrowsPrior > decreaseAmount ? accountBorrowsPrior - decreaseAmount : 0;* On top of that, if a position is partially/fully liquidated before the repayment, the repayer will also simply donate the repayAmount to the contract.
Recommendations	Consider refunding any excess repayment amount. This will require refactoring of the function.
Comments / Resolution	Acknowledged, see comment issue #15

Issue_19	Difference in interest rate accrual due to compounding nature
Severity	**Medium**
Description	The totalBorrows variable is updated whenever the accrue modifier is invoked. Due to the additive nature of the totalBorrows variable and the fact that the interest is always based on the previous totalBorrows variable, a more frequent update will result in an overall higher interest rate. Generally speaking, interest/funding rates should not be manipulatable by more/less frequent updates.
Recommendations	Consider if this issue is desired to be fixed. In that case, the whole interest rate calculation must be refactored.
Comments / Resolution	Acknowledged.

Issue_20	Lack of safety threshold can result in immediate liquidation post-borrow
Severity	**Medium**
Description	Whenever a user borrows tokens on behalf of a tokenId, the following check is executed at the end of the function which ensures that the position is not liquidatable based on the new state: *if(borrowAmount > repayAmount) require(* *ICollateral(collateral).canBorrow(tokenId, address(this), accountBorrows),* *"ImpermaxV3Borrowable: INSUFFICIENT_LIQUIDITY"* *);* Since canBorrow in fact only checks if a liquidation is liquidatable, it

	can happen that a user is liquidated immediately in the subsequent block either due to interest accumulation or a small price change. A similar issue is existent within the redemption mechanism in the collateral module.
Recommendations	Consider implementing a safety threshold similar to all other lending protocols which does not allow borrowing up to the full LTV.
Comments / Resolution	Acknowledged, see comment issue #15

Issue_21	exchangeRate is updated if interest is not paid back
Severity	**Low**
Description	The exchangeRate determines the underlying value of a PoolToken. This topic has been thoroughly elaborated within (Appendix: Exchange Rate). Since the totalBorrows variable is constantly increased (due to interest), the exchangeRate is increased even if the accrued interest has not yet been collected. In itself, that doesn't seem like a problem. However, under certain circumstances, this issue can amplify existing vulnerabilities.
Recommendations	A change of this logic would require a refactoring of the whole accrual mechanism.
Comments / Resolution	Acknowledged.

TokenizedV3 Module

ImpermaxERC721

The ImpermaxERC721 contract is a customized version of the ERC721 contract. It is used as the underlying token for the TokenizedUniswapV3Position as well as ImpermaxV3Collateral contract.

It exposes all necessary ERC721 functions such as:

- approve
- setApprovalForAll
- transferFrom
- safeTransferFrom

Furthermore, it enables permit-based approvals.

Privileged Functions

- none

Issue_22	Violation of ERC721 specs
Severity	**Medium**
Description	The ImpermaxERC721 contract is a rewritten form of the ERC721 contract. Due to some changes in the code, this inherently violates the ERC721 spec in certain spots: https://eips.ethereum.org/EIPS/eip-721#specification For example, ownerOf is a mapping here while it should be a function: *function ownerOf(uint256 tokenId) public view virtual returns (address) {* *return _requireOwned(tokenId);* *}* Thus, it does not revert if a tokenId is not owned.
Recommendations	Consider following the ERC712 specs. It has to be noted that this means adjustment throughout the borrow and collateral module. Thus another option might be to acknowledge it and carefully document this change for third-party users.
Comments / Resolution	Resolved.

Issue_23	Revert of isApprovedForAll does not automatically remove tokenId approval
Severity	**Medium**
Description	The contract allows anyone where isApprovedForAll is true to approve a tokenId towards any address: *require(auth == address(0) || auth == owner || isApprovedForAll[owner][auth], "ImpermaxERC721:INVALID_APPROVER");* *getApproved[tokenId] = to;* If this global approval is removed, this does not remove any potentially previous approval settings for unique tokenIds which have been made by the address with the previous global allowance. This is prone to errors as the original owner most likely will not check any unique approval made by the address with the global allowance.
Recommendations	Consider removing the ability to approve tokens by addresses with a global allowance.
Comments / Resolution	Acknowledged.

Issue_24	Assignment of nonce to tokenId may be suboptimal
Severity	**Low**
Description	Currently, permit is using a nonce which is assigned to a tokenId. If a user for example signs a message for a specific tokenId with a very loose deadline, it can happen that the user transfers this tokenId out to another user and at some point in time will get the tokenId back, which means the signed message would still be valid. In the scenario where the nonce is tied to the user address instead of the tokenId, it would be possible for the user to mark the signed message as invalid in the meantime. However, with the current setup, it is not possible.
Recommendations	Consider if it makes sense to tie the nonce to the user address or keep it tied to the tokenId. However, if the frontend signs messages with a reasonable deadline, it should be sufficient to acknowledge this issue.
Comments / Resolution	Acknowledged.

TokenizedUniswapV3Position

The TokenizedUniswapV3Position contract is a liquidity wrapper contract which mints a NFT which is representing a unique pool position. Each position has the following traits:

- fee
- tickLower
- tickUpper
- liquidity
- feeGrowthInside0LastX128
- feeGrowthInside1LastX128
- unclaimedFees0
- unclaimedFees1

Besides of tokenizing a position, the contract offers the following other features:

a) Split: A position owner can split a tokenId and create a new tokenId which proportionally moves liquidity and fees to the new tokenId

b) Join: A position owner can join a tokenId to an already existing tokenId which increases the liquidity and fees by applying the liquidity weighted average of feeGrowthInsideLast from the old tokenId and the new tokenId

c) Reinvest: This function allows the permissionless auto-compounding of fees into a position's liquidity. The caller of this function receives a small share of the fees as caller incentive.

Appendix: Liquidity Weighted Average

The join function merges one tokenId into another tokenId. Since it is possible that these two tokenIds have a different feeGrowthInsideLastX128, a specific formula is applied which preserves the total outstanding fees. More specifically, the liquidity weighted average is used to calculate the new feeGrowthInsideLast based on the merged liquidity. The formula for this is as follows:

> ((feeGrowthLast * liquidityTokenId) + (feeGrowthLast * liquidityTokenIdToJoin)) / newLiquidity

This methodology preserves outstanding fees.

Appendix: Fee Growth calculation

(This appendix is based on Algebra but translates 1:1 to UniswapV3)

In Algebra, pools accumulate fees during swaps. If a swap from X -> Y is executed, the pool accumulates Y, if the swap is vice-versa, the pool accumulates X. Since v3 usually has different positions over different ranges, it is not possible to simply accrue these fees over the full range. Instead, fees must be accrued by these positions which are currently in range.

To accommodate that, a totalFeeGrowthToken0/1 variable was implemented for tokenX/tokenY. This variable tracks the fee per liquidity, scaled by 1x128.

> totalFeeGrowthToken += fees * (1<<128) / liquidity

Example:

A swap has happened and accrued 1e18 fees in tokenX while the active liquidity is 1100e18:

totalFeeGrowthToken += 1e18 * (1<<128) / 1100e18
totalFeeGrowthToken = 3.039e35

Now another swap happens which accrues 10e18 fees in tokenX while the activeLiquidity is 1000e18

totalFeeGrowthToken = 3.039e35
3.039e35 += 10e18 (*1<<128) / 1000e18
3.039e35 += 3.402e36
totalFeeGrowthToken = 3.705e36

Now since we are aware that totalFeeGrowthToken aggregates the fee per unit of liquidity, the following calculation allows us to determine how much fees a specific position has accrued:

> (innerFeeGrowthToken - innerFeeGrowthTokenLast) * liquidity

Now as we can see, the variable which is used includes “inner”. We need to understand that, as already explained, each v3 position incorporates its very own storage mechanism.

Here are a all important properties:

a. Whenever a position is newly added, it consists of upper and lower tick

   i. If the tick had liquidity before, outerFeeGrowth is not adjusted

   ii. If the tick had no liquidity before and the tick is <= currentTick, outerFeeGrowth is set to totalFeeGrowth

   iii. If the tick had no liquidity before and the tick is >= currentTick, outerFeeGrowth is not set

b. Whenever a position is newly added, feeGrowthInside will be set to the current feeGrowthInside for the range

c. Whenever a swap happens and a tick for a position is crossed, the outerFeeGrowth of the tick will be set to [feeGrowthTotal - outerFeeGrowth]

d. outerFeeGrowth always corresponds to the feeGrowth outside of the position to either the left or the right side (depending on lower/upper)

e. Depending on the currentTick and the range, the following calculations are used to calculate feeGrowthInside:

   i. **currentTick < lowerTick** (lowerTick was crossed; X > Y):
   > innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth

   ii. **currentTick >= upperTick** (upperTick was crossed; Y > X):
   > innerFeeGrowth = upper.outerFeeGrowth - lower.outerFeeGrowth

iii. **lower < current = upperTick** (upperTick was crossed; Y > X)
> innerFeeGrowth = upper.outerFeeGrowth - lower.outerFeeGrowth

iv. **lowerTick < currentTick < upperTick** (no tick was crossed):
>innerFeeGrowth = totalFeeGrowth - lower.outerFeeGrowth - upper.outerFeeGrowth

v. **lowerTick = currentTick** < upperTick (lowerTick was not crossed):
> innerFeeGrowth = totalFeeGrowth - lower.outerFeeGrowth - upper.outerFeeGrowth

f. The last scenario: lowerTick = currentTick < upperTick is a special scenario, as one would assume the calculation must be as follows:

> innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth.

The reason is because one would assume that is that the scenario:

lowerTick = currentTick

resulted in a cross of the tick and thus lower.outerFeeGrowth = [feeGrowthTotal - outerFeeGrowth]. This is the only thing that would make sense.
However, in fact, in that scenario, the tick has not crossed. This is because of the edge-case within SwapCalculation._calculateSwap for the zeroToOne scenario where currentTick = nextTick - 1. Consider the following example: lowerTick = 0; currentTick = 10. X -> Y swap is being executed nextTick = 1, which means the lowerTick is not crossed. However, currentTick will be set to nextTick - 1 which means now that the condition: currentTick = lowerTick is met without the tick being crossed and thus tick = 0 still remains the same outerFeeGrowth as before and the calculation must be executed as if the currentTick is inside the range.

g. The calculation explicitly assumes underflows/overflows which is derived from the "unchecked" setting. There are specific scenarios where this is necessary to keep the integrity of the fee calculation mechanism. For example, in the following situation, this becomes relevant: https://github.com/Uniswap/v3-core/issues/573.

More specifically, the following conditions must be true:
a) currentTick is lower than lowerTick
b) lowerTick of the position was not yet initialized
c) upperTick was initialized. If now a position is added based on these properties, innerFeeGrowth will be calculated as follows:

> innerFeeGrowth = lower.outerFeeGrowth - upper.outerFeeGrowth
This means it becomes negative.

Example:
a) currentTick = 0
b) lowerTick = 1
c) upperTick = 10
d) lowerTick.outerFeeGrowth = 0
e) upperTick.outerFeeGrowth = 100
f) totalFeeGrowth = 100

This will result in: innerFeeGrowth = 0 — 100

Appendix: newFeeGrowthInsideLastX128

In a Uniswap V3—like system, each position tracks:

- **Liquidity**: how large the position is in that range.
- **feeGrowthInsideLast**: the "fee growth" accumulator checkpoint at the time it was last updated.
- **unclaimedFees**: the fees not yet collected by the position.

When additional fees accrue, the global fee accumulators (e.g., feeGrowthInside) increase. The position ultimately claims:

(feeGrowthInsidecurrent−feeGrowthInsideLast) × liquidity + unclaimedFees

Suppose we merge two positions (call them A and B). Each has its own:

- liqA; feeGrowthLastA
- liqB; feeGrowthLastB

and some unclaimed fees on the side.

After merging, we have:

1. **New liquidity:** newLiquidity=liqA+liqB
2. **Combined unclaimed fees:** newUnclaimedFees=(unclaimedA+unclaimedB)
3. A single checkpoint **newFeeGrowthInsideLast** that must preserve the same final outcome as if the positions had been collected separately.

To calculate newFeeGrowthInsideLast:

We define:

tA = feeGrowthLastA×liqA,

tB = feeGrowthLastB×liqB

newFeeGrowthInsideLast = tA * tB / newLiquidity

The newFeeGrowthInsideLast variable essentially yields the weighted average feeGrowthLast for both positions.

Core Invariants:

INV 1: Users can only redeem/split tokenIds which they have approvals for or own

INV 2: The liquidity provision and mint call must happen in the same transaction

INV 3: Join must preserve the unclaimed tokenX/tokenY amount for both tokenIds

INV 4: Any tokens which are not used for autocompounding must be aggregated into unclaimedFees0/1 mapping

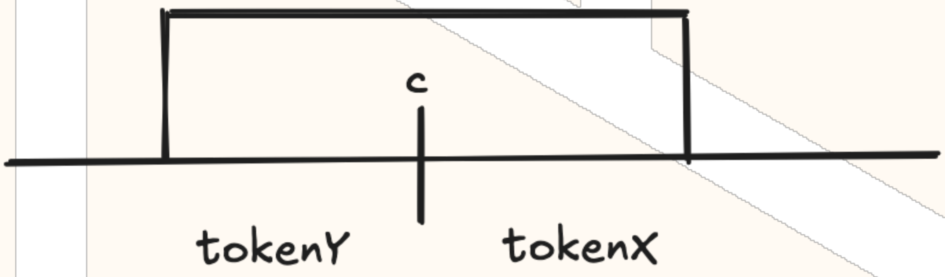
INV 6: Split is only callable if tokenId is owned or approval granted

INV 7: Join is only possible for tokenIds with the same configuration

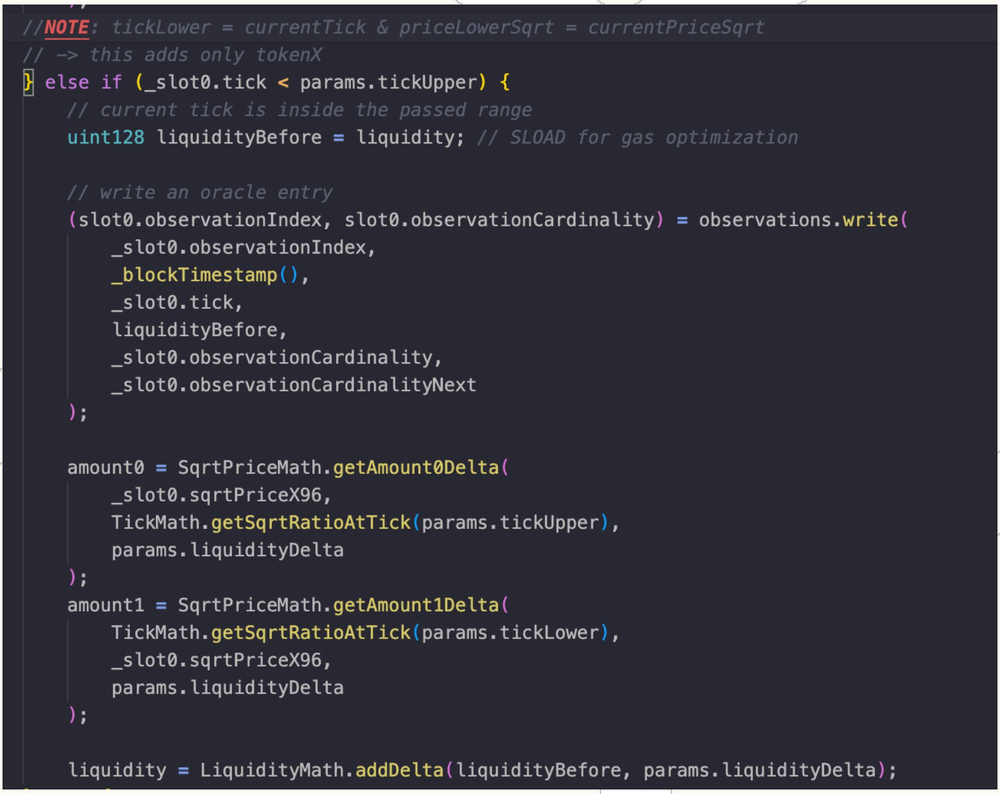
Privileged Functions

- none

Issue_25	Governance: Fees can be taken by malicious autocompounder
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, via a malicious autocompounder, it is possible to steal fees from a position or prevent reinvesting.
Recommendations	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior.
Comments / Resolution	Acknowledged.

Issue_26	Manipulation of oracle price by pushing malicious pool
Severity	**High**
Description	Uniswap uses the following TWAP oracle: https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L16 This oracle returns in the known manner the arithmeticMeanTick: https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L34 On top of that, it returns the harmonicMeanLiquidity: https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L40 The same function is used in the current ImpermaxV3 architecture. The harmonicMeanLiquidity is obviously a safeguard to prevent low liquidity pairs from having a large impact on the TWAP. Otherwise users could simply create empty pairs and swap the price to a desired price to impact the TWAP. The background behind the harmonicMeanLiquidity is that it uses the liquidity from the pair. In UniswapV3, liquidity is only added if the currentTick is in b/w the range. This can be illustrated as follows: 

In the code, this is displayed as follows:



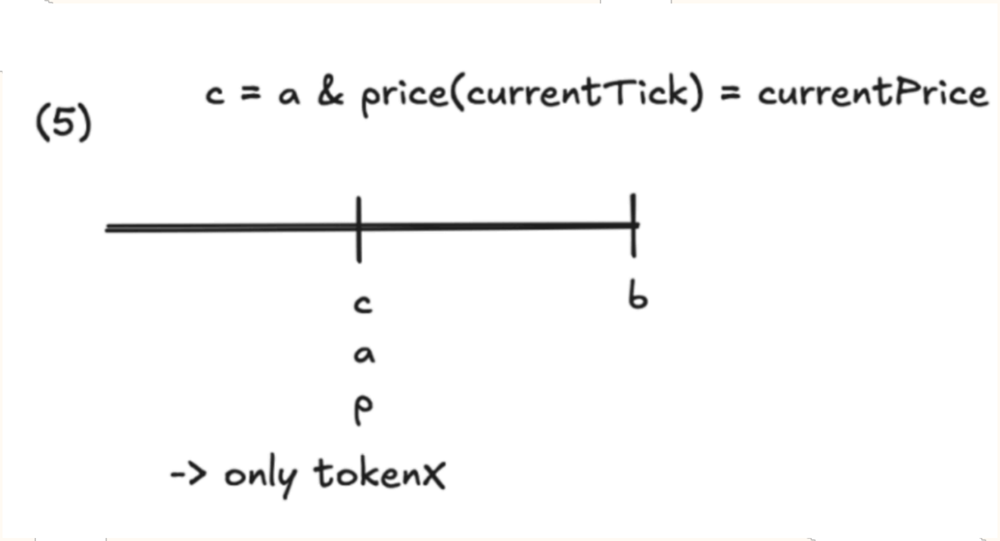
```
//NOTE: tickLower = currentTick & priceLowerSqrt = currentPriceSqrt
// -> this adds only tokenX
} else if (_slot0.tick < params.tickUpper) {
    // current tick is inside the passed range
    uint128 liquidityBefore = liquidity; // SLOAD for gas optimization

    // write an oracle entry
    (slot0.observationIndex, slot0.observationCardinality) = observations.write(
        _slot0.observationIndex,
        _blockTimestamp(),
        _slot0.tick,
        liquidityBefore,
        _slot0.observationCardinality,
        _slot0.observationCardinalityNext
    );

    amount0 = SqrtPriceMath.getAmount0Delta(
        _slot0.sqrtPriceX96,
        TickMath.getSqrtRatioAtTick(params.tickUpper),
        params.liquidityDelta
    );
    amount1 = SqrtPriceMath.getAmount1Delta(
        TickMath.getSqrtRatioAtTick(params.tickLower),
        _slot0.sqrtPriceX96,
        params.liquidityDelta
    );

    liquidity = LiquidityMath.addDelta(liquidityBefore, params.liquidityDelta);
```

Summarized that means liquidity is increased even if we have only tokenX, as it will display the following (valid) scenario:



	This is a (not-so-known) but possible edge-case for UniswapV3. In itself, this doesn't seem to be an issue for the code because if you swap to the left side, you automatically cross the tick and it decreases the liquidity again. However, UniswapV3 uses an oracle which is not only based on the average tick but also the average liquidity (https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L1) in fact, the consult function returns the following values: > arithmeticMeanTick > harmonicMeanLiquidity These values can then be used within the getWeightedArithmeticMeanTick function (https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L140) which gives a higher weight to these ticks with higher liquidity. This is a standard mechanism to prevent the impact of manipulated pairs with low liquidity. However, due to the already explained state above, it is now possible to exploit this safety mechanism, which essentially renders it useless and can be abused manipulate the price of a pool (specifically, if external protocols attempt to integrate pools which allow for all fee tiers, an exploiter can then just initialize the pool with a new fee tier, executing this exploit which gives massive weight to the manipulated tick, resulting in a loss of funds depending on the oracle usage. To reiterate, the following conditions must be fulfilled: a.Swap an empty pool to a tick where: i) tokenX is very expensive ii) sqrtPriceX96 is equal to the currentTick b.Add liquidity with tickLower = currentTick and tickUpper any tick

	i) This has the effect that you only add tokenX and since tokenX is sitting at a large price, nobody is going to execute MEV bc nobody is going to buy out the tokenX which you have added You have now successfully: a. Brought the tick to a large price b. Large liquidity If you now continue writing to the oracle (via adding the same range e.g.) you will write invalid states to the oracle which can then be abused as explained. This will then allow to drain the protocol by depositing the token with the large value and borrowing the token with the low value
Recommendations	Consider not allowing the permissionless addition of new pools.
Comments / Resolution	Resolution 1: The contract has been refactored to a large degree. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved

Issue_27	DoS due to addition of pool with insufficient lookback period
Severity	**High**
Description	Currently, the getPool function allows for adding any of the three fee tier pools to the poolsList. This has not only the goal to activate a pool for deposits but also uses the pool address for oracle purposes. A malicious user can create a pool with a new fee tier which has insufficient lookback period (< 1800 sec), which then results in a revert of the oracle consultation. The reason why this is marked as high risk is due to the fact as the protocol will be essentially shut down which also means liquidations are impossible.
Recommendations	Consider not allowing the permissionless addition of new pools.
Comments / Resolution	Resolution 1: The contract has been refactored to a large degree. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved.

Issue_28	FeeGrowth logic within UniswapV3 expects underflow/overflow possibility
Severity	**High**

Description	Within the Appendix: FeeGrowth calculation, we have already explained an important property which is related to underflows/overflows.
	
	The feeGrowth logic calculation within UniswapV3 allows for underflow within the calculation:
	
	*function getFeeGrowthInside(
	mapping(int24 => Tick.Info) storage self,
	int24 tickLower,
	int24 tickUpper,
	int24 tickCurrent,
	uint256 feeGrowthGlobal0X128,
	uint256 feeGrowthGlobal1X128
	) internal view returns (uint256 feeGrowthInside0X128, uint256 feeGrowthInside1X128) {
	Info storage lower = self[tickLower];
	Info storage upper = self[tickUpper];
	
	// calculate fee growth below
	uint256 feeGrowthBelow0X128;
	uint256 feeGrowthBelow1X128;
	if (tickCurrent >= tickLower) {
	feeGrowthBelow0X128 = lower.feeGrowthOutside0X128;
	feeGrowthBelow1X128 = lower.feeGrowthOutside1X128;
	} else {
	feeGrowthBelow0X128 = feeGrowthGlobal0X128 - lower.feeGrowthOutside0X128;
	feeGrowthBelow1X128 = feeGrowthGlobal1X128 - lower.feeGrowthOutside1X128;
	}
	
	// calculate fee growth above
	uint256 feeGrowthAbove0X128;
	uint256 feeGrowthAbove1X128;*

```
        if (tickCurrent < tickUpper) {
        feeGrowthAbove0X128 = upper.feeGrowthOutside0X128;
        feeGrowthAbove1X128 = upper.feeGrowthOutside1X128;
        } else {
        feeGrowthAbove0X128 = feeGrowthGlobal0X128 -
upper.feeGrowthOutside0X128;
        feeGrowthAbove1X128 = feeGrowthGlobal1X128 -
upper.feeGrowthOutside1X128;
        }

        feeGrowthInside0X128 = feeGrowthGlobal0X128 -
feeGrowthBelow0X128 - feeGrowthAbove0X128;
        feeGrowthInside1X128 = feeGrowthGlobal1X128 -
feeGrowthBelow1X128 - feeGrowthAbove1X128;
        }
```

This means, feeGrowthInside can underflow while at the same time, the calculation will still keep its integrity. A specific example has been included in the appendix.

This is also explicitly handled within the UniswapV3 NFPM, as it allows for underflow here as well:

```
        position.tokensOwed0 += uint128(
        FullMath.mulDiv(
        feeGrowthInside0LastX128 - position.feeGrowthInside0LastX128,
        position.liquidity,
        FixedPoint128.Q128
        )
        );
        position.tokensOwed1 += uint128(
        FullMath.mulDiv(
        feeGrowthInside1LastX128 - position.feeGrowthInside1LastX128,
        position.liquidity,
        FixedPoint128.Q128
        )
        );
```

	However, the current implementation of the contract does not incorporate that scenario:
	
	`function _getfeeCollectedAndGrowth(Position memory position, address pool) internal view returns (uint256 fg0, uint256 fg1, uint256 feeCollected0, uint256 feeCollected1) {`
	`bytes32 hash = UniswapV3Position.getHash(address(this), position.tickLower, position.tickUpper);`
	`(,fg0, fg1,,) = IUniswapV3Pool(pool).positions(hash);`
	
	`uint256 delta0 = fg0 > position.feeGrowthInside0LastX128 ? fg0 - position.feeGrowthInside0LastX128 : 0;`
	`uint256 delta1 = fg1 > position.feeGrowthInside1LastX128 ? fg1 - position.feeGrowthInside1LastX128 : 0;`
	
	`feeCollected0 = delta0.mul(position.liquidity).div(Q128).add(position.unclaimedFees0);`
	`feeCollected1 = delta1.mul(position.liquidity).div(Q128).add(position.unclaimedFees1);`
	`}`
	
	Moreover, using the calculation within the join function can result in an erroneous outcome if one of both positions has a negative feeGrowth as that would essentially break the fee logic for the newly merged position

Recommendations	The calculation must be refactored to account for this edge-case, it should be allowed to underflow similar as in the NFPM. Furthermore, we recommend the following changes to the join function: a) Implement permission checks towards the “tokenToJoin” b) Compound both positions to ensure feeGrowthInsideLast corresponds to the same value
Comments / Resolution	Resolution 1: The join function now calculates the leftover fee and simply executes an addition. Moreover, feeGrowthInsideLast is set to the latest value for the corresponding pool. However, no zero burn is executed which can

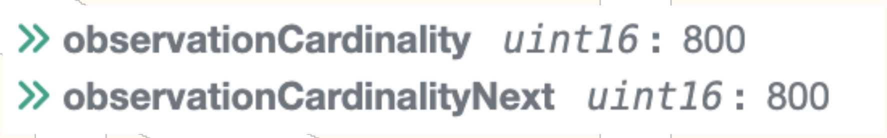
	result in feeGrowthInsideLast not being the actual value which is tied to the pool. We could not inspect any immediate side-effects from that since the new position can claim these remaining fees in the same way as both old isolated positions could have done. However, it would be considered as best-practice to do that. The contract has been refactored to a large degree. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved, however, we still recommend executing a zero burn beforehand to ensure no unexpected side-effects occur and in an effort to simply update the final position with the very latest feeGrowthInside value.

Issue_29	DoS due to lack of cardinality check
Severity	**High**

Description	The UniswapV3 TWAP oracle stores historic data via a circular buffer, which means if the end is reached, it will override the first data point from the beginning again.
	
	By default, the cardinality for this buffer is 1:
	
	*/// @notice Initialize the oracle array by writing the first slot. Called once for the lifecycle of the observations array
	/// @param self The stored oracle array
	/// @param time The time of the oracle initialization, via block.timestamp truncated to uint32
	/// @return cardinality The number of populated elements in the oracle array
	/// @return cardinalityNext The new length of the oracle array, independent of population
	function initialize(Observation[65535] storage self, uint32 time)
	internal
	returns (uint16 cardinality, uint16 cardinalityNext)
	{
	self[0] = Observation({
	blockTimestamp: time,
	tickCumulative: 0,
	secondsPerLiquidityCumulativeX128: 0,
	initialized: true
	});
	return (1, 1);
	}*
	
	It can then be increased via this function:
	
	function increaseObservationCardinalityNext(uint16 observationCardinalityNext)

```
        external
        override
        lock
        noDelegateCall
        {
        uint16 observationCardinalityNextOld =
slot0.observationCardinalityNext; // for the event
        uint16 observationCardinalityNextNew =
        observations.grow(observationCardinalityNextOld,
observationCardinalityNext);
        slot0.observationCardinalityNext =
observationCardinalityNextNew;
        if (observationCardinalityNextOld !=
observationCardinalityNextNew)
        emit
IncreaseObservationCardinalityNext(observationCardinalityNextOld,
observationCardinalityNextNew);
        }
```

For most pools, the cardinality is not very high (even if they are live for a long time). For example, the USDC/WETH pool on base has a cardinality of 800:



```
>> observationCardinality   uint16 : 800
>> observationCardinalityNext   uint16 : 800
```

This allows a clever attacker to update the cardinality once per block within a 1800 seconds interval such that the end of the buffer is reached and the most historic data point is overridden. This will then result in a similar revert as in the above mentioned issue, as the most historic lookback period has been overridden and now the most historic lookback period is less than 1800 seconds in the past.

	The reason why this is marked as high risk is due to the fact as the protocol will be essentially shut down which also means liquidations are impossible.
Recommendations	Consider ensuring that the cardinality is at least 1800 for each pool which is added towards the poolList.
Comments / Resolution	Resolution 1: The contract has been refactored to a large degree. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved.

Issue_30	Reinvest will always result in position becoming less healthy
Severity	**Medium**
Description	The reinvest function compounds any unclaimed fees into the position by simply adding liquidity via the autocompounder contract. This practice will always result in the position becoming more unhealthy due to two factors: a) The more obvious factor is the fee which is taken by the autocompounder, as it will nominally reduce the tokenX/Y amounts which are entitled to the position b) The less obvious factor is the fact that the fee flows nominal into the value calculation of the position. This means LOWEST/HIGHEST will not have an impact on the ratio of the fee as it is always considered as nominal tokenX and nominal tokenY amount. if now however the fee is added to the liquidity of the pair, the next value calculation via LOWEST/HIGHEST will now (depending of the range and the safetyMargin) result in a change of the ratio against the favor of the position. If for example the position had tokenX entitled as fee and this tokenX amount is now added to the liquidity, the next HIGHEST application will result in tokenX being "shifted" to tokenY (depending on the position's range and safetyMargin), which then, combined with the fact that this tokenY is now worth less due to the price increase results in a decrease of the value. (The HIGHEST application essentially swaps X to Y using the curve price of the position, while HIGHEST is potentially higher as the experienced curve price which then results in a nominal value loss) Additionally to mention but less impactful, reinvest can be DoS'd if a user with single sided fee attempts to claim and another user frontruns this call with a swap such that the claimer's range is single sided towards the counter side.

	In such a scenario where positions already accumulated a lot of funds and the acModule was set to address(0) and is now suddenly changed to the correct implementation, this issue could even be considered as a high risk issue instead of a medium risk issue.
Recommendations	Consider removing the permissionless nature of this function.
Comments / Resolution	Resolution 1: Within the getPositionData function, 5% of the fee is pessimistically deducted and applied on the values for LOWEST/HIGHEST. This means positions are considered to be liquidated earlier than they should in an effort to prevent unexpected side-effects from reinvesting. In our opinion, this is not the most elegant solution as user's will always suffer from that. Instead, we recommend reverting that change and allowing only for reinvesting if the caller is the actual owner and the tokenId is not used for collateral, which aligns with our initial recommendation of removing the permissionless nature of this function. This will not only limit user flexibility but also decreases potential state transitions which can often be the root-cause of exploits.It will certainly increase the security of the overall system. The contract has been refactored to a large degree. This must be fully audited in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. Resolution 2: Resolved , it has to be noted that this will have the final impact that positions with outstanding fees are considered as liquidatable earlier.

Issue_31	Possible DoS for specific pairs due to safecast within getPositionData
Severity	**Medium**
Description	The getPositionData function crafts all three prices as follows: *priceSqrtX96 = oraclePriceSqrtX96();* *uint160 currentPrice = safe160(priceSqrtX96);* *uint160 lowestPrice = safe160(priceSqrtX96.mul(1e18).div(safetyMarginSqrt));* *uint160 highestPrice = safe160(priceSqrtX96.mul(safetyMarginSqrt).div(1e18));* It is possible that for a pair with different decimals and a high price, the price of the pair can reach up to the maximum allowed price in UniswapV3: 1461446703485210103287273052203988822378723970342 This could be for example: 1460446703485210103287273052203988822378723970342 which is a valid price and works with UniswapV3 pairs. If now, the safetyMarginSqrt is 1.1e18, the HIGHEST price will be: 1606491373833731113616000357424387704616596367376 which is above uint160.max and thus reverts. More specifically, if tokenX is a token with 6 decimals and tokenY is a token with 21 decimals, a value of 1 will already result in a price of 1e15 which translates into 2505414483750479286512002635546469086 If now the price is higher than 1, the possibility for overflow is given which results in a revert. This issue is inherent existing due to the uint160 enforcement of the

	LiquidityAmounts library, which means it is theoretically possible that a pair aligns with UniswapV3 by having a sqrtPriceX96 below uint160 but due to the safetyMarginSqrt then exceeds it. The same counts for the LOWEST price.
Recommendations	Consider carefully selecting pairs which are going to be added.
Comments / Resolution	Acknowledged.

Issue_32	Loss of fee accumulation if position is not immediately deposited
Severity	**Low**
Description	First of all, it is clear that a user must invoke the mint function in the same transaction as the position is being created on behalf of the TokenizedUniswapV3Position contract, otherwise another user could “skim” the position and steal it. Besides that, which is a known design choice, there is also another issue, if the user does not mint in the same transaction. Even if no other user skims the position and the initial depositor mints at another point in time while receiving the corresponding tokenId, an issue can arise where the user loses out fees b/w [minting; lastPoolUpdate]. The root-cause of this issue is the fact that feeGrowthInside0/1LastX128 is being set to the last feeGrowth value of the pool. If the user has now minted a position and deposits only at a later point, all rewards for (liquidity * (feeGrowthNow - feeGrowthCreation) will be permanently lost because fee can only be collected based on each representative tokenId’s liquidity and the delta between the current feeGrowth of the pool and the stored feeGrowthInsideLast for the position, which is now lost due to update of feeGrowthInsideLast to the

	pools feeGrowth whenever the tokenId is minted.
Recommendations	Consider ensuring that mint is always called in the same transaction as the liquidity is added.
Comments / Resolution	Acknowledged.

Issue_33	Possible zero liquidity positions
Severity	**Low**
Description	The mint function as well as the split function can result in zero liquidity tokenIds. This can result in unexpected edge-cases.
Recommendations	Consider at least checking for mint that liquidity is non-zero.
Comments / Resolution	Acknowledged.

Collateral Module

CStorage

The CStorage contract exposes the storage layout for the collateral module.

Privileged Functions

- none

No issues found.

CSetter

The CSetter contract handles all governance functions which allow the setting of important parameters, such as:

- safetyMarginSqrt
- liquidationIncentive
- liquidationFee

Privileged Functions

- _setSafetyMarginSqrt
- _setLiquidationIncentive
- _setLiquidationFee

Issue_34	Governance: Setting of parameters
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, if the safetyMarginSqrt variable is increased, positions become immediately liquidated earlier or an increase of liquidationPenalty can have a negative impact on the post-liquidation position health factor in case only a part of the debt is repaid.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

ImpermaxV3Collateral

The ImpermaxV3Collateral contract allows users to deposit their TokenizedUniswapV3Position tokenIds into this contract which can then be used as collateral to borrow the corresponding underlying tokens.

Whenever a user has deposited a tokenId, the user receives the corresponding tokenId in the form of the ImpermaxV3Collateral ERC721 token.

This tokenId can then be fully or partially redeemed as long as it remains sufficiently collateralized. The contract furthermore exposes the seize function which is invoked upon the liquidation call and the restructureBadDebt function which forgives bad-debt from a position in order to make it liquidatable.

The collateralization check depends on the position value within the TokenizedUniswapV3Position contract and the borrowed amounts which are tied to this tokenId within the ImpermaxV3Borrowable contracts.

Appendix: Liquidation Process:

Any position which is considered as liquidatable (see Appendix: Liquidation Determination) can be fully/partially liquidated by anyone. The liquidator must simply call the liquidate function on the corresponding ImpermaxV3Borrowable contract to repay the tied underlying token (if the position has debt in that token).

This flow will then automatically invoke the seize function which calculates the repayToCollateralRatio (see appendix) and ensures the subsequent split does not exceed 1e18, as otherwise the liquidation call would revert.

It will then calculate the seizePercentage:

repayToCollateralRatio * liquidationIncentive
and the feePercentage

> (repayToCollateralRatio * liquidationFee / (1e18 - seizePercentage)

While splitting the corresponding shares of the tokenId to the liquidator and to the reservesManager

Appendix: repayToCollateralRatio

Whenever a liquidation is happening, the repayToCollateral value is determined which represents the nominal percentage of the repayment value proportionally to the collateral value.

> repayToCollateralRatio = value(repayAmount) * 1e18 / value(collateral)

If the collateral value is 100e18 and the repay value is 80e18, repayToCollateralRatio becomes 0.8e18.

Appendix: Liquidation Incentive

To properly incentivize users to execute a liquidation, the liquidationIncentive variable is introduced which is used as multiplier on repayToCollateralRatio to then determine how many % a user receives based on his repayment amount

> repayToCollateralRatio = value(repayAmount) * 1e18 / value(collateral)

> seizePercentage = (value(repayAmount) * 1e18 / value(collateral)) * liquidationIncentive / 1e18

Similar to that, a liquidationFee towards the protocol is taken:

> (repayToCollateralRatio * liquidationFee / (1e18 - seizePercentage)

The collateral will thus not only be decreased by the proportional repaid amount but also by the liquidationIncentive and the liquidationFee.

Core Invariants:

INV 1: TokenizedUniswapV3Position tokenId transfer and mint must be called in the same transaction

INV 2: Redeem is only callable if tokenId is owned or approval granted

INV 3: Must not be able to seize position with bad debt

INV 4: liquidationPenalty must flow into bad-debt calculation

INV 5: The tokenId must not be liquidatable post-redeem

INV 6: Value of repaymentAmount multiplied by liquidationPenalty must always be smaller than the value of the position

INV 7: The liquidationIncentive must be applied on repayToCollateralRatio

INV 8: A position must be fully liquidatable after the debt is restructured

INV 9: underlying must point towards the corresponding TokenizedUniswapV3Position contract

INV 9: Partial liquidations must make the position more healthy

INV 10: A tokenId can only be fully redeemed if there is no debt tied to it

INV 11: A tokenId can only be redeemed as long as it remains sufficiently collateralized after the redemption

INV 12: A position must be liquidatable after bad debt has been restructured

Privileged Functions

- none

Issue_35	Liquidations will never work in case of symmetric position borrowing
Severity	**High**

Description	A position is always considered as liquidatable whenever:
	
	value(debt) * penalty > value(collateral)
	
	while value is determined based on HIGHEST/LOWEST price
	
	In the scenario of symmetric borrowing (collateral tokenX; borrow tokenX), liquidations never work because once a position is considered as liquidatable:
	
	value(debt) * penalty > value(collateral)
	
	It is at the same time considered as being in bad debt:
	
	value(debt) * penalty > value(collateral)
	
	(since bad-debt is always related to current price)
	
	This can be illustrated based on a trivial example:
	
	collateralX = 100e18
	debtX = 98e18
	LTV = 0.98
	
	The liquidatable calculation is based on the LOWEST/HIGHEST price which doesn't make a difference in the nominal value, because the debt and borrow tokens are both tokenX:
	
	LOWEST = 0.5:
	
	collateralXValue = 50e18

	debtXValue = 49e18
	LTV = 0.98
	
	HIGHEST = 1.5:
	
	collateralXValue = 150e18
	debtXValue = 147
	LTV = 0.98
	
	This means, a user would need to call restructureBadDebt before a liquidation and when this is called, the position will be restructured which means some bad debt is forgiven but at the same time once it is successfully restructured, the position is again in a position where it is considered as liquidatable but in bad-debt at the same time, which results in an infinite loop which makes the position impossible to liquidate.
	
	On top of that, an attacker can leverage this flaw to not only prevent the liquidation of the own position per-se but also benefit from it by obtaining the PoolToken for tokenX, redeeming before bad-debt is restructured and gain virtual value (which is then forgiven and paid by all other lenders). This can be repetitively used to syphon value out of the system.
	
	Similar to that, if safetyMargin is 1e18, liquidations are impossible as well.

Recommendations	Consider allowing the liquidation of a position in the case where it has been restructured in the same block. Eventual side-effects of such a change should be considered.
Comments / Resolution	Partially resolved. A blockOfLastRestructureOrLiquidation mapping has been incorporated which is set to the current block.number if a restructuring has just happened. Within the liquidation call, a short-circuit path was implemented which allows for liquidating a position if it has been just restructured in the same block.

	It has to be noted that this mapping is also set within the liquidation call which then allows for iterative partial liquidations in the same block. However, this introduces potential side-effects where a partial liquidation results in a position to become less healthy and eventually enter bad-debt. In that scenario, a position can become immediately liquidated without the need to restructure bad debt beforehand. Therefore, we recommend removing this setting and simply set block.number to zero again after the liquidate call. Furthermore, if a partial liquidation would result in a position becoming healthy again, it could be further liquidated in the same block, which doesn't seem desirable. **Resolution 2**: Fully resolved, the client indicated that it is a design choice to allow multiple subsequent partial liquidations even if that means the position would become more healthy in between, liquidations are still allowed. It has to be noted that if a position becomes underwater for any reason after a partial liquidation (if for example the penalty is very high), a full liquidation would still result in a revert due to the following line (because an attempt would be made to pay more value back than in fact is collateralized): *require(repayToCollateralRatio.mul(liquidationPenalty()) <= 1e36, "ImpermaxV3Collateral: UNEXPECTED_RATIO");*

Issue_36	Bad debt is incorrectly forgiven if reserveManager is address(0)
Severity	**High**
Description	The restructureBadDebt function forgives the bad debt based on the assumption that liquidationPenalty consists of liquidationIncentive and liquidationFee: *uint postLiquidationCollateralRatio = positionObject.getPostLiquidationCollateralRatio();* > collateralValue * 1e18 / (collateralNeeded * liquidationPenalty) In the scenario where the reserveManager is address(0), this will not take the liquidationFee from a user's position upon liquidation: *if (liquidationFee > 0 && reservesManager != address(0)) {* An important invariant of the protocol is that positions should get fully liquidated post-restructuring if the full debt is being repaid but in that scenario, the user will still have a share of the position left while all the debt is repaid. This can be nicely illustrated with a position which already accrued bad-debt above the collateral: a) Position is in bad-debt - tokenXCollateral = 100e18 - tokenXDebt = 100.1e18 b) Position is being restructured (bad debt is forgiven) - tokenXCollateral = 100e18 - tokenXDebt = 96.25e18 c) Position gets liquidated but reserveManager = address(0) - seizePercentage = 98.175e18

	- no liquidation fee since reserveManager = 0 d) Post liq state: - tokenXCollateral = 1.825e18 - tokenXDebt = 0 The user was effectively fully liquidated but still has a share of the position left. Under specific circumstances, such as if the user has a lending position which collects the interest before the restructuring, the user can even syphon some funds illegally out of the system. (If he wasn't beforehand liquidated on point)
Recommendations	Consider not allowing reserveManager to be address(0).
Comments / Resolution	Resolved.

Issue_37	restructureBadDebt can be front run by users to profit from beneficial ER
Severity	**Low**
Description	The restructureBadDebt function decreases the ER because it essentially forgives bad-debt by virtually repaying it (without actually transferring the tokens in). This will decrease the ER. A clever user can frontrun such a call to benefit from the old ER and can immediately deposit afterwards again to bypass any loss.
Recommendations	The only solution for this issue is to implement a time-based deposit/withdrawal limitation.
Comments / Resolution	Acknowledged.

Issue_38	Partial liquidation may result in position becoming less healthy
Severity	**Low**

Description	An important invariant in lending protocols is that partial liquidations should always result in positions becoming more healthy. This can be simply explained via proportions:
	
	Collateral = 100
	Debt = 80
	LTV = 0.8
	
	repay 20
	
	Collateral = 80
	Debt = 60
	LTV = 0.75
	
	This invariant is violated in case the allowed LTV is very high and the position is near its allowed LTV plus experiences a penalty during liquidation:
	
	Collateral = 100
	Debt = 90
	LTV = 0.9
	
	repay 10 + 15% penalty
	
	Collateral = 88.5
	Debt = 80
	LTV = 0.9039
	
	This issue is inherent present for all lending protocols where a penalty is applied to the collateral.

Recommendations	Consider being reasonable with the liquidationPenalty for pools with a low safetyMargin.
Comments / Resolution	Acknowledged.

Issue_39	Lack of onERC721Received function prevents safeTransferFrom
Severity	**Low**
Description	Since the collateral module expects to accept ERC721 tokens tied to the TokenizedUniswapV3Position contract, it is expected that the onERC721Received function is exposed. That is however not the case, which renders transfers via safeTransferFrom impossible.
Recommendations	Consider implementing the aforementioned function.
Comments / Resolution	Resolved.

Issue_40	Zero redemption possibility
Severity	**Low**
Description	The redeem function currently does not ensure that percentage != 0. This can lead to unexpected scenarios if users accidentally or intentionally call redeem with percentage = 0.
Recommendations	Consider implementing a sanity check for this scenario.
Comments / Resolution	Acknowledged.

V3Math

CollateralMath

After the audit, the precision was changed. This was not included in the audited part. We recommend a careful check for any potential side-effects.

The CollateralMath contract is a fundamental part of the ImpermaxV3 ecosystem as it essentially determines when a position is considered as liquidatable and when a position is in bad debt.

This is done via the isLiquidatable and isUnderwater functions.

Appendix: Liquidation Determination

The isLiquidatable function is the most fundamental function to check whether a tokenId and the corresponding debt is considered as undercollateralized. This function is considered whenever a user wants to redeem a tokenId, an liquidation attempt is executed or further funds are borrowed.

This function basically considers

a) The debt value (including the penalty)

b) The collateral value

Based on the PRICE.LOWEST and PRICE.HIGHEST the value of the debt and collateral is calculated which then determines a position as liquidatable once debtValue > collateralValue

More specifically, this is done as follows (both paths are checked upon isLiquidatable)

Price.LOWEST:

a) Calculate debt value from borrowedX/borrowedY based on LOWEST price
b) Calculate collateral value from lowestX/lowestY based on LOWEST price
c) Check if debtValue > collateralValue

Price.HIGHEST:

a) Calculate debt value from borrowedX/borrowedY based on HIGHEST price
b) Calculate collateral value from highestX/highestY based on HIGHEST price
c) Check if debtValue > collateralValue

Value of a position is calculated as follows:

(amountX * sqrtPrice / 2**96) + (amountY / sqrtPrice / 2**96)

It is important to mention that any unclaimed fees from the collateral position will stay in tokenX/tokenY and will not be impacted by the ratio change of LOWEST/HIGHEST price for the liquidity.

The LOWEST/HIGHEST is related to the safetyMargin and the corresponding LTV and will be evaluated within the Appendix below.

Appendix: safetyMargin

The positionObject struct represents a tokenId's position. It is built as follows:

- realXYs
- priceSqrtX96
- debtX
- debtY
- liquidationPenalty
- safetyMarginSqrt

The main crux within this struct is the realXYs value as it not only represent the amount of tokenX/tokenY based on the current price but also the amount based on currentPrice * safetyMargin and currentPrice / safetyMargin. It is important to understand that the safetyMarginSqrt is applied on the sqrtPrice and not the nominal price.

This will yield three different values for tokenX/tokenY based on:

- LOWEST
- HIGHEST
- sqrtPrice

This methodology acts as a “variable LTV” which is derived from the ImpermaxV2 protocol.

The whitepaper can be found here:
https://www.impermax.finance/_files/ugd/5c39d7_4fe2a5f2b65e44998d4d39c4da98a7b9.pdf (3.3. Collateralization Model). Indeed, since a UniswapV3 position around the current price is not different from a UniswapV2 position, this model applies to all UniswapV3 positions as well, with the edge-cases where the price goes out of the position range and the position only consists of tokenX or tokenY.

In that scenario, the LTV will largely deviate and a position can become liquidated way earlier / later based on the fact if the borrow is symmetric (collateral tokenX, borrow tokenX; liquidate later) or asymmetric (collateral tokenX, borrow tokenY; liquidate earlier).

One can summarize this core concept in simple terms by seeing the safetyMargin as LTV.

The higher the safetyMargin, the lower the LTV. This means if a user has provided collateral to the system and safetyMargin is high, the LOWEST/HIGHEST prices, which are actually used to calculate the position value against the favor of the user (see Appendix: Liquidation Determination) deviate more from the current price and thus the user will get liquidated earlier. The formula for this is:

LTV = (safetyMarginSqrt / 1e18) ** 2 / liquidationPenalty * 1e18

On the other hand, if the safetyMargin is zero, LTV is 100% (ignoring the penalty) which means LOWEST/HIGHEST = current price which means the user can borrow up to his whole tokenX/tokenY collateral.

There are two notable edge-cases:

a) If the user has collateral in only one token (even after the safetyMargin was applied on the price) and at the same time borrows only the same token, the LTV can be up to 100% (excluding the penalty)

b) Any unclaimed fee does not flow into the liquidity and will thus not experience a ratio shift based on LOWEST/HIGHEST but will nominally flow into the value calculation.

Summarized, the LTV can variate from 1/safetyMargin up to 100% (in the scenario of a symmetric borrow scenario).

Appendix: Collateral Ratio

The Collateral Ratio determines whether a position is in bad-debt and is calculates as follows:

> collateralValue(currentPrice) * 1e18 / (debtValue(currentPrice) * liquidationPenalty / 1e18

This ratio is always used within restructureBadDebt to determine whether the position must be restructured or not.

Privileged Functions

- none

No issues found

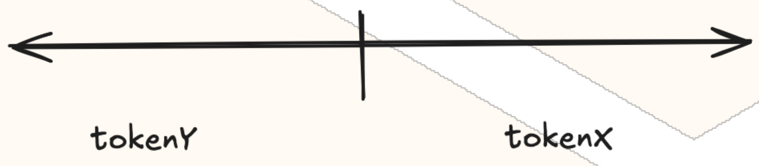
UniswapV3CollateralMath

The UniswapV3CollateralMath contract is responsible for returning the tokenX/tokenY amount based on a certain range and price. This library mimics the known LiquidityAmounts library

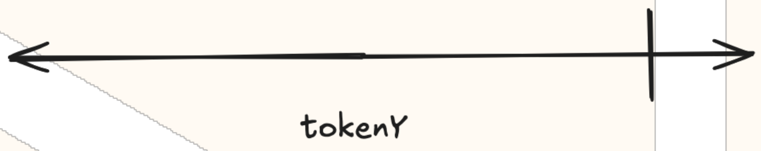
(see Algebra example:
https://github.com/cryptoalgebra/Algebra/blob/e1f358c1c2c9f6cda455c6f0934b333af624e964/src/core/contracts/test/LiquidityAmounts.sol)

In UniswapV3, liquidity is provided over a range of ticks and based on the currentPrice, the ratio of tokens are provided.

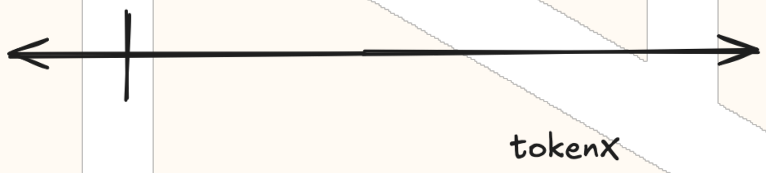
A simple illustration of an equal range:



A simple illustration once the price has increased:



A simple illustration once the price has decreased:



The key takeaway is that liquidity above the currentPrice always consists of tokenX while liquidity below the currentPrice consists of tokenY. If liquidity is spread across the range (currentPrice is crossed), it consists of tokenX/tokenY

The math behind the liquidity to token conversions is as follows:

amountX = liquidity * (sqrtUpper - lowerBoundSqrt) / (sqrtUpper * lowerBoundSqrt) * 2**96

amountY = liquidity * (upperBoundSqrt - lowerPriceSqrt) / 2**96

The UniswapV3CollateralMath library has formed equivalent calculations:

amountX = (liquidity *2**96 / lowerBoundSqrt) - (liquidity *2**96 / upperPriceSqrt)

amountY = (liquidity * upperBoundSqrt / 2**96) - (liquidity * lowerPriceSqrt / 2**96)

Privileged Functions

- none

No issues found

UniswapV3Position

The UniswapV3Position library is responsible for fetching the hash of a position which is derived as follows:

> keccak256(owner, tickLower, tickUpper)

This hash is then used to fetch the data for a corresponding position on a UniswapV3Pool:

mapping(bytes32 => Position.Info) public override positions;

Privileged Functions

- none

No issues found.

V3Oracle

UniswapV3WeightedOracleLibrary

The UniswapV3WeightedOracleLibrary contract is forked from the original Uniswap OracleLibrary contract:

(https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L140)

It exposes the original consult function which is responsible for consulting the oracle to then return the arithmeticMeanTick as well as the harmonicMeanLiquidity.

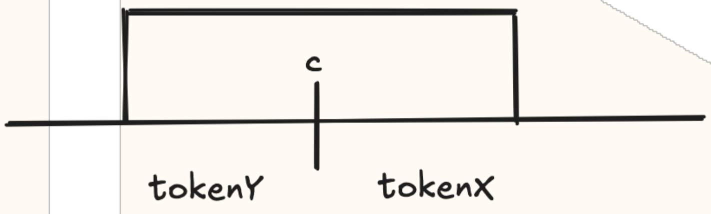
These are then used by the getArithmeticMeanTickWeightedByLiquidity function which essentially commingles different arithmeticMeanTicks from the different pools and emphasizes tick values from pools with higher liquidity by calculating the weighted average:

arithmeticMeanWeightedTick = ((harmonicMeanLiqiudityPool1 * arithmeticMeanTickPool1) + (harmonicMeanLiquidityPool2 * arithmeticMeanTickPool2) + ...) / (harmonicMeanLiquidityPool1 + harmonicMeanLiquidityPool2 + ...)

Appendix: Active Liquidity

In UniswapV3, not every range addition will automatically increase the liquidity. In fact, liquidity is only increased if the currentTick is inside a newly added range. Otherwise, liquidity will not be increased. Moreover, the liquidity change during swaps is handled as follows:

UniswapV3 implements a liquidity net which derives from the upper and lower ticks of a position:



Once a position is added around the currentTick with a liquidity of 500e18, this will write the following state:

- liquidity = 500e18
- bottom.liquidityDelta = 500e18
- top.liquidityDelta = -500e18

If a swap from Y to X is executed and this swap brings the currentTick outside of the range, top.liquidityDelta will be added towards liquidity, bringing it to zero. If now a reverse swap from X to Y is executed while bringing the currentTick again in the range, top.liquidityDelta will be removed from the liquidity (which makes it positive, since its negative).

Privileged Functions

- none

No issues found.

UniswapV3OracleTwap

The UniswapV3OracleTwap contract exposes the oraclePriceSqrtX96 function which is invoked by the TokenizedUniswapV3Position.getPositionData function and consults an array of pools with the goal to receive the arithmeticMeanWeightedTick between all pools.

Privileged Functions

- none

No issues found.

ImpermaxV3OracleChainlink

The ImpermaxV3OracleChainlink is an oracle adapter which fetches two token prices from two Chainlink oracles and then calculates the price0 in price1. Examples:

token0 = ETH and token1 = USDT
> return the ETH price in USDT

token0 = ETH and token1 = AVAX
> return the ETH price in AVAX

It is important that both token prices are consulted in USD, otherwise the token0InToken1Price will be erroneous since it would be based on the wrong denomination.

Governance can add new tokens and their corresponding USD price feeds via the _addTokenSources function, while an additional safety measurement is the verifiyTokenSource state variable which can be turned on and off and while it is turned on, the _addTokenSources function executes further sanity checks for the to be added price feeds.

On top of that, the contract implements an optional fallback oracle which can be consulted if one of both feeds returns an invalid price

Appendix: Decimal normalization

The oraclePriceSqrtX96 function consults both chainlink oracles and returns their provided values while at the same time it is possible that both tokens have different decimal values.

Since token0InToken1Price returns the token1 amount for 1 wei of token0 while at the same time it is possible that token0 has 6 decimals and token1 has 18 decimals. If the price is simply 1, *it would result in 1e12 since 1 wei of token0 results in 1e12 of token1 due to the decimal difference of 12*. The same can be applied vice-versa which results in 1e-12.

Furthermore, (while most oracles return the price with 8 decimals) it is possible for some oracles to return a price with more or less than 8 decimals. In that scenario, the decimal deviation must be incorporated as well into the calculation.

Thus, the Impermax team developed the following approach to receive the sqrtX96Price of token0 in token1:

a) Calculate the total decimals of for token0 and the oracle of token0 (same for token1):

   > totalDecimalsToken0 = token0.decimals + oracle0.decimals

b) calculate the delta decimals:

   > deltaDecimals = totalDecimalsToken0 - totalDecimalsToken1

c) Calculate sqrtPriceX96

   > sqrtPriceX96 = sqrt(price0 * (2**128) / price1) * (2**32)

d) Calculate absolute delta decimals and scale x96 by the decimals (in a looped fashion):

   > deltaDecimalsAbs = abs(deltaDecimals)

   > scaleX96 = (2**96) * 10^deltaDecimalsAbs

e) Multiply scaledX96 with (2**48) and take the sqrt

   > scaleX96 = sqrt(scaleX96 * (2**48))

f) Scale priceSqrtX96 up or down depending on sign of deltaDecimals

   DOWN

   > priceSqrtX96 = priceSqrtX96 * (2**96) / scaleX96

   UP

   > priceSqrtX96 = priceSqrtX96 * scaleX96 / (2**96)

Privileged Functions

- _acceptAdmin
- _setPendingAdmin
- _setVerifyTokenSource
- _setFallbackOracle
- _addTokenSources

Issue_41	Potential side-effects due to real price and pair price deviation
Severity	**Medium**
Description	The contract uses the Chainlink oracle as source of truth which is expected to always return the correct price. If the pair deviates from the real price, under all our illustrated scenarios, it means that users provide more nominal value while the collateral value is lower. This is inherent b/c any deviation will result in a "swap" which requires the swapper to provide tokenX at a higher price than the current price to get out tokenX (or vice-versa). If for example the current price is 1 and a user provides 50x; 50y which grants the user a collateral value of 100, now another user adds liquidity but the pair deviates to 1.1 which means the user needs to add 104 tokenY as liquidity for the same liquidity amount while the real price is still 1 and displays a collateral value of 100 while the user in fact has provided a collateral value of 104 **Scenario 1:** - real price = 1.1 - pair price = 1 - Alice adds 50 tokenX, 50 tokenY to the vault - nominal value in tokenY: 50 * 1.1 + 50 = 105 tokenY - Since real price is used to calculate ratio for pool: - Alice's provided liquidity is determined as 104 tokenY at a price of 1.1

	Alice effectively provided more value than what is reflected **Scenario 2:** - pair price = 1.1 - real price = 1 - Alice adds 104 tokenY to the vault - nominal value in tokenX: 104 / 1.0 - 104 tokenX - Since real price is used to calculate ratio for pool: - Alice's provided liquidity is determined as 50 tokenX and 50 tokenY at a price of 1 - Alice effectively provided 104 tokenX while the pair shows a value of 50 tokenX and 50 tokenY Alice effectively provided more value than what is reflected We will now also explain the root-cause of why such a deviation always results in more provided collateral than reflected. For that, we will pick the first example: - real price = 1.1 - pair price = 1 - Alice adds 50 tokenX, 50 tokenY to the pair b/w [0.909; 1.1] At this point, the collateral is worth (in tokenY): 50 * 1.1 + 50 = 105 If now the oracle calculates the ratio from the pair as if the price would be 1.1: 104 tokenY Now we need to understand that Alice has provided all her tokenX b/w

	[1.0 ; 1.1]. Due to the fact that the real price is 1.1, it will illustrate “as if a swap has happened” on the curve of 1.0 to 1.1, which gives an average selling price of tokenX for tokenY which is lower than 1.1. However, in the initial illustrated example, we have calculated tokenX value with 1.1 (due to the real price being 1.1). However, since the user has only added the liquidity b/w 1.0 and 1.1, the swap effectively resulted in less tokenY for the provided tokenX.
Recommendations	To prevent any unforeseen edge-cases, we strongly recommend adding a deviation check within the oraclePriceSqrtX96 function in the TokenizedUniswapV3Position contract which reverts if the deviation from the pair’s sqrtPriceX96 to the real price is larger than a specific threshold. The deviation can be determined by governance and even completely disabled if one is certain that such a deviation does not introduce edge-cases. This is an important safeguard which we always recommend for all protocols which deal with value calculations for concentrated liquidity protocols and indeed most liquidity management protocols expose this safeguard.
Comments / Resolution	Acknowledged.

Issue_42	Side-effects due to truncation
Severity	**Low**

Description	The oraclePriceSqrtX96 function executes multiple actions which result in truncation. In certain scenarios, this can amend the final output price and result in a deviation.
	
	Consider the following two examples which should yield the same price but in fact are different:
	
	Scenario 1:
	
	token0Price = 1
	token0Decimals = 6
	oracle0Decimals = 8
	oracle0Price = 100000000
	
	token1Price = 3000
	token1Decimals = 18
	oracle1Decimals = 8
	oracle1Price = 300000000000
	
	priceSqrtX96 = 144650172662492649560526028800000
	
	Scenario 2:
	
	token0Price = 1
	token0Decimals = 6
	oracle0Decimals = 18
	oracle0Price = 1000000000000000000
	
	token1Price = 3000
	token1Decimals = 18
	oracle1Decimals = 8
	oracle1Price = 300000000000
	
	priceSqrtX96 = 1446501726624926496477138649088000

	As one can see, even though the price should be exactly the same, the return value slightly deviates because in the second scenario the oracle decimals are denominated with 1e18 instead of 1e8 which results in a discrepancy due to truncation.
Recommendations	Consider further elaborating potential scenarios and determine whether the impact is acceptable. We recommend intense fuzzing for that purpose.
Comments / Resolution	Acknowledged.

Issue_43	Lack of basic return value checks
Severity	**Low**
Description	The oraclePriceSqrtX96 function in the ImpermaxV3OracleChainlink contract exhibits several integration issues that could lead to potential vulnerabilities: 1. **Lack of Sequencer Uptime Check:** The contract does not verify the operational status of the underlying blockchain sequencer or network. This omission could lead to price data being processed during periods of network instability or outages. 2. **Lack of Staleness Check:** There is no mechanism to ensure that the price data retrieved from the oracles is recent. Without checking for data staleness, the contract may act on outdated price information, potentially resulting in incorrect pricing calculations. 3. **Lack of Deviation Check:** The implementation does not validate whether the newly reported price deviates significantly from previous price data.

	This missing safeguard increases the risk of manipulation or sudden, anomalous price shifts affecting the computed oracle price. 4. **Lack of Fallback Return Value Validation:** In scenarios where the primary price feeds return non-positive values, the contract defers to a fallback oracle. However, it does not perform additional validation on the fallback oracle's return value, potentially propagating erroneous or manipulated data. Overall, these standard issues with oracle integrations could lead to scenarios where the contract relies on unreliable or manipulated price feeds, thereby increasing the risk of an incorrect price
Recommendations	Consider whether it makes sense to implement these mentioned safeguards. However, generally speaking, Chainlink oracles are usually deemed as safe.
Comments / Resolution	Acknowledged.

Issue_44	Immutable feeds
Severity	**Low**
Description	The contract's design enforces immutability on the oracle sources for each token via the tokenSources mapping. Once an oracle source is set for a token through the _addTokenSources function, it cannot be updated or removed, as subsequent calls require that the token's source is zero (i.e., not yet initialized). This is explicitly expected as per NATSPEC but can result in unexpected scenarios.
Recommendations	Consider if it makes sense to increase the flexibility in that regards.
Comments / Resolution	Acknowledged.

Issue_45	Potential overflow revert within oraclePriceSqrtX96 function
Severity	**Low**
Description	The contract exposes a risk of potential overflow revert if price0 is a very large value: *priceSqrtX96 = Math.sqrt(uint256(price0).mul(Q128).div(uint256(price1))).mul(Q32);* This could result in an overflow during the multiplication with (2**128).
Recommendations	Consider ensuring that no exotic tokens will be used.
Comments / Resolution	Acknowledged.

Issue_46	Math formula can be simplified
Severity	**Informational**
Description	The oraclePriceSqrtX96 function calculates the sqrtPriceX96 for token0 in token1, adjusted by decimals. The math for this function is redundantly complicated and can be simplified in multiple areas. For example: > sqrt(price0 * (2**128) / price1) * (2**32) is similar to: > sqrt(price0 * (2**192) / price1) And likely a way to prevent potential overflows Furthermore, the looping: *for (uint i = 0; i < deltaDecimalsAbs; i++) {* *scaleX96 = scaleX96.mul(10);* *}* can just be simplified to: scaleX96 = (2**96) * 10^deltaDecimalsAbs
Recommendations	We do not recommend changing the math formula but instead simply adding comments which illustrate the simplified formula such that third-parties can grasp it easier.
Comments / Resolution	Acknowledged.