Impermax V3 Core Re-Audit Report

Prepared by:

Charles Wang, Owen Thurm, Riley Holterhus

June 30, 2025

Contents

1 Introduction

1.1 About Impermax

Impermax is a lending protocol focused on liquidity provider tokens, allowing users to borrow against positions from automated market makers such as Uniswap. Lenders receive interest payments, while borrowers can gain leveraged exposure using their LP tokens as collateral. For more information, visit impermax.finance.

1.2 About the Auditors

Charles Wang, Owen Thurm, and Riley Holterhus are experienced smart contract auditors who operate through their own auditing firms or work independently. For this engagement, the three collaborated to conduct a follow-up review of the Impermax V3 core contracts. The review was conducted in response to a security incident that occurred in April 2025, with a focus on assessing the changes introduced after the incident and reviewing for additional issues.

1.3 Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the authors have made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an “as-is” basis and DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM.

2 Audit Overview

2.1 Scope of Work

The scope of this review can be found in Impermax's impermax-v3-core GitHub repository, specifically starting with commit hash 4a217adc110b0e5dab895b825b2f0e74755513fc. The audit encompassed the following core contracts:

- contracts/libraries/CollateralMath.sol
- contracts/extensions/ImpermaxV3OracleChainlink.sol
- contracts/extensions/TokenizedUniswapV3Position.sol
- contracts/BAllowance.sol
- contracts/BSetter.sol
- contracts/BStorage.sol
- contracts/CSetter.sol
- contracts/CStorage.sol
- contracts/ImpermaxV3Borrowable.sol
- contracts/ImpermaxV3Collateral.sol
- contracts/PoolToken.sol

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "High", "Medium", or "Low". These severities aim to capture the impact and likelihood of each potential issue.

In total, **45 findings** were identified. This includes **4 high**, **15 medium**, and **26 low** severity findings. The status of each issue, along with the responses from the Impermax team, can be found in the findings section below.

3 Findings

3.1 High Severity Findings

3.1.1 Blacklisted addresses can prevent liquidations

Description: A change in the code has refactored `split()` to transfer any unclaimed fees of the token being split directly to the owner of the corresponding collateral NFT. Since `split()` is called during liquidations, it's important that this transfer never reverts to block the liquidation process. This can become a problem when dealing with tokens such as USDC and USDT, which implement blacklists that cause `transfer()` to revert when sending to a blacklisted address.

For example, an attacker who owns an address on the USDC blacklist could exploit this by setting up a USDC-related Impermax position using a fresh wallet, then transferring their collateral NFT to their blacklisted address. This would make the position unliquidatable, as the transfer of unclaimed fees would revert during liquidation. A griefer could also abuse this behavior by donating their collateral NFT to a blacklisted address, even if they do not control it.

Recommendation: To ensure liquidations cannot be blocked, consider modifying the `collect()` call to send fees to the `TokenizedUniswapV3Position` contract itself, and track unclaimed fee balances internally so that users can claim their tokens later on.

Impermax: Resolved with commit 48d5072 and commit bcb1106.

Auditors: Verified. The code now tracks unclaimed fees per token, and when a `split()` occurs, all unclaimed fees are attributed to the original token. These fees are not transferred during the split anymore, and can be claimed later via `claim()`.

3.1.2 Dangerous approval pattern

Description: In the borrow function the `msg.sender` is validated with the `_checkBorrowAllowance()` function. However as stated in the comments, the `borrow()` function is intended to be called by a router contract.

Therefore the user is expected to approve the router contract to make borrows on their behalf before initiating the borrow. The approval must be performed in a multicall within the same transaction as the borrow is carried out, and ensuring that the borrow amount is reduced to zero. Otherwise a malicious actor could backrun the approval and frontrun the borrow initiated by the user to use the router to borrow funds against the victim's NFT and send them to their own malicious recipient.

Otherwise, the router must implement specific validations to ensure that only the owner of the NFTLp in the collateral contract may invoke the borrow action.

Recommendation: Either refactor the authentication at the borrow function level, or be sure to implement strict validations in the router such that addresses which do not own or are not approved for the borrowing against the tokenId cannot initiate a borrow through the router.

Otherwise ensure that all interactions with the router are in a multicall and do not leave any dangling approvals, and be sure to warn integrations about this.

Impermax: We're already using both the recommended safeguards in this case:

1) the router executes its own validations check
2) the permit style approval is used to ensure that the router is approved to borrow only within the same transaction

Auditors: Acknowledged.

3.1.3 Configuration updates may cause bad debt

Description: The `_setLiquidationIncentive()` and `_setLiquidationFee()` functions will immediately affect the under water status of all positions. In some cases when raising either one of these configurations positions may immediately become under water.

With significant increases in the liquidation incentive and liquidation fee it may be possible for a malicious actor to frontrun the configuration updates and open a position up to the limit of being liquidatable and/or under water, which then becomes immediately underwater after the configuration update.

This is particularly clear with positions which use a collateral LP entirely in one token, and a borrow balance entirely in the same token, whereby they can borrow directly up to the limit of being under water.

Recommendation: Be sure to use a private RPC to make collateralization configuration updates. Furthermore be sure to monitor all existing positions to ensure that none of them become liquidatable or under water with the planned configuration update with `_setSafetyMarginSqrt()`, `_setLiquidationIncentive()`, or `_setLiquidationFee()`.

Impermax: Addressed in commit 068682a.

Auditors: Verified. The code has been updated so that `safetyMarginSqrt`, `liquidationIncentive`, and `liquidationFee` are initialized to their maximum values to begin with, and the admin is allowed to lower these parameters but cannot increase them beyond their current values at any time.

3.1.4 `blockOfLastRestructureOrLiquidation` can be abused

Description: The `blockOfLastRestructureOrLiquidation` logic allows for liquidation one position multiple times in the same block - even if it is in underwater or non-liquidatable.

Under most scenarios, this is not an issue. However, we found a specific scenario where an attacker can exploit this and extract more from a position than he should be able to.

Consider the following scenario: Alice notices that here position is unhealthy as follows:

- collateral = 50e18
- debt = 40.1e18

- LTV = 0.8

Now Alice wants to increase her collateral by using the join function and adding 10e18 tokens to a tokenID and at the same time borrowing 5e18 tokens. It would now make the position healthy again

- collateral = 60e18
- debt = 45.1e18

Bob notices this action and knows he has the possibility of frontrunning Alice's top-up by repaying the full debt. However, Bob is clever and keeps in mind that the can liquidate more than once in the same block. Hence Bob sandwiches Alice's top-up, first he liquidates a part, then Alice tops up which makes her position healthy again. Bob now backruns the top-up and he is able to fully repay the 45.1e18 debt instead of only the 40e18 debt and thus he received more fees for doing so.

Recommendation: A fix for this issue is non-trivial.

Impermax: We will update the UI in such a way to make it impossible to increase the debt on a liquidatable position. I think no change is needed to the smart contract given the high unlikelihood of this attack.

Auditors: Acknowledged.

3.2 Medium Severity Findings

3.2.1 ClaimFees can create bad debt

Description: The claim fees function can be invoked to immediately remove a position's fee value from its collateral while in use. This combined with the behavior of `blockOfLastRestructureOrLiquidation` allows for the creation of bad debt which can selectively be levied on one group of lenders or another.

The high level attack flow is as follows:

- Do a 1 wei liquidation to write the lastLiquidationBlock, this effectively keeps the numbers the same but allows us to liquidate even with an underwater position.
- Claim fees from the position
- The position is now under water, but can still be liquidated
- Liquidate the entire loaned amount through borrowable A
- Restructure debt for the loaned amount from borrowable B

Recommendation: Firstly, the claiming of fees should not allow collateral value to be immediately reduced when there is an active loan, at least to below the underwater threshold. To resolve this, an under water check could be added to the claimFees function to ensure that positions are not under water after claiming fees.

Secondly, it should not be possible to liquidate a position which is under water. To resolve this consider removing the `blockOfLastRestructureOrLiquidation` logic in the seize function.

Impermax: Resolved in commit 05f1369.

Auditors: Verified. Unclaimed fees no longer contribute to a position's collateral value in any circumstance.

3.2.2 Lack of oracle change possibility

Description: Currently, it is not possible to change the oracle address once it is set. This can result in issues where it is required to switch to a new oracle or block important functions by removing the oracle.

Recommendation: Consider allowing to change the oracle.

Impermax: By being able of changing the oracle, a malicious admin would be able to steal most of the protocol TVL. This in my opinion is a bigger concern, which is why the oracle is immutable.

Auditors: Acknowledged.

3.2.3 Excess liquidator repayment is donated

Description: In the `liquidate()` function the `repayAmount` is minimized to be at maximum the borrow balance of the user, however the liquidate function expects that the repay amount has been transferred to the contract before the liquidation is carried out, therefore when the `repayAmount` is capped it will likely not match the amount sent. This is likely to happen often given that it's hard to predict what the exact borrow balance of a user will be when a transaction is ultimately recorded. As a result, excess tokens will have often been sent by liquidators, and is allowed by the `balance.sub(totalBalance) >= repayAmount` validation. However these excess tokens are not used to repay any debt nor are returned to the liquidator.

Recommendation: Consider returning any excess tokens to the liquidator in the liquidate function.

Impermax: Liquidator bots fix this by checking the latest borrow balance at the beginning of the transaction, before calling `liquidate()`.

Auditors: Acknowledged.

3.2.4 Excess user repayment is donated

Description: In the `_updateBorrow()` function if the repayment amount is greater than the account's borrows the `accountBorrows` is assigned to 0 and the excess repayment amount is ignored.

In the event that a user sends an excess repayment this amount will be donated to the borrowable pool. It may be common for a user to send excess repayment tokens since it is not clear how much the user's debt will be at the time of the transaction being recorded.

Recommendation: Consider refunding excess repayment amounts to the user at the end of the borrow function.

Impermax: Same resolution as "Excess liquidator repayment is donated". In this case the router does the check.

Auditors: Acknowledged.

3.2.5 Deviation threshold affects liquidators

Description: Each aggregator feed can be off up to its deviation threshold, most often the deviation threshold for a Chainlink aggregator is 50 or 100 basis points. Therefore in a reasonable worst case scenario a reported price can

be off by up to 2% (the maximum deviation for both asset's feeds).

The direction of the inaccuracy can either go in the favor of liquidators or against the favor of liquidators. The case where a position is made up of entirely token1 with a debt entirely of token0 maximizes the affect of this deviation.

In the case where the inaccuracy goes in the favor of liquidators and against the borrower, this can leave a position less healthy after partial liquidation as measured by the fair value, without any inaccuracy due to deviation.

In the case where the inaccuracy goes against the favor of liquidators, they may choose to not liquidate a position if the inaccuracy overshadows the gain from the liquidation incentive. In this case positions which are reportedly liquidatable by the protocol may remain without being liquidated, increasing risk of bad debt.

Recommendation: Be aware of the potential for deviation inaccuracy in either direction, and consider the deviation threshold of the feeds for a market when configuring the liquidation parameters.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.2.6 `repayToCollateralRatio` allows unpayable bad debt

Description: In the seize function it is validated that the `repayToCollateralRatio` multiplied by the `liquidationPenalty` will not exceed 100%. However it is allowed to be equal to 100%, allowing for a 100% split of the existing NFT being seized. In the case where the NFTLp is made underwater within a block, this can lead to a position which has its entire collateral value split out and only debt remaining. If the NFTLp tokenId was the only token in that range this can create a temporarily un-restructurable position with the following steps:

- Assume liquidationFee is set to zero for simplicity
- Position with tokenId0 is liquidatable
- Do a 1 wei seize to update the block of last restructure or liquidation
- A Chainlink aggregator update or liquidation configuration makes the position immediately under water within the same block
- A second seize where the repayToCollateralRatio * liquidationIncentive = 100%
- Position with tokenId1 is created with all of the collateral value
- Liquidator redeems tokenId1, this is possible because there is no debt for tokenId1
- tokenId0 has no liquidity associated with its collateral, but still holds all of the bad debt
- Restructure now fails for tokenId0 fails when attempting to update the fees with burn due to this check: https://github.com/Uniswap/v3-core/blob/c5ccf4d28a73fde90f0bb9ea3fd299d7d2bcdf83/contracts/libraries/Position.sol#L54

This can be resolved by minting 1 wei to the same tick range as tokenId0, however this gives ample time for lenders to exit the vault before the bad debt is manually realized, and for the underwater position to continue to accrue bad debt before this is realized and fixed.

Recommendation: Be aware of this edge case and have a mechanism to add liquidity to the range so that the position is restructurable and liquidatable within a short period of time.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.2.7 Lack of pausing mechanism in emergency mechanism

Description: Currently, it is not possible to pause the contract in emergency situations. This has significant downside in the scenario of unexpected events.

Recommendation: Consider implementing a pausing mechanism

Impermax: As with finding "Lack of oracle change possibility", we'd rather not give too much powers to the admin.

Auditors: Acknowledged.

3.2.8 Symmetric borrowing allows for creating bad debt

Description: Whenever the same token is deposited and borrowed, the LTV is near 100% (depending on incentive + penalty).

This means within the next block, a position is considered as liquidatable and underwater. This opens up an attack-vector where the attacker can be the majority lender and depositor/borrower at the same time, accruing interest in the very next block while withdrawing the lending position right before restructuring. Then the attacker can restructure the bad debt and essentially stole interest. This can be amplified if the attacker is MEV proficient as he can decide to not restructure/liquidate until another user submits such a transaction and then frontrun it. This gives the attacker more time to accrue and steal interest.

Recommendation: Consider making `restructureBadDebt()` permissionless and implementing a safety threshold to ensure borrowing is not allowed up to the liquidation point - 1.

Impermax: I think this attack is high risk-low reward.

I've ran some number, and I think by doing this a lender could increase his APR by 10-20% in the best scenario.

The problem is that from the moment the borrow position is opened, liquidation bots will try to liquidate it. Even if the attacker is MEV proficient he won't be able to keep this kind of position opened for long. And if he loses the MEV competition and gets liquidated, he will lose much more than what he could possibly gain.

Auditors: Acknowledged. We warn that there may be unconsidered scenarios relating to permissionless debt socializing, and believe this is a risk that should be kept in mind.

3.2.9 Leveraging of threshold borrowing

Description: Currently, it is possible to borrow up to the liquidation treshold - 1. This means, a position can be considered as liquidatable in the very next block (due to interest), if the price remains the same.

In itself, this is already an issue. However, this can also be weaponized by a malicious user to increase the chance of creating bad-debt and tricking the system. Multiple other issues have been created which require the pre-requisite of borrowing up to the threshold. Eliminating this possibility will prevent most of these issues.

Recommendation: Consider implementing a safety threshold.

Impermax: We made this design choice for 2 reasons:

1) Given our unique collateralization model, setting a max_borrowable_threshold lower than the liquidation_thershold is not trivial.
2) This design is still safe for the lenders. If a user decides to open a position with the max LTV available, he will be immediately liquidated.

Auditors: Acknowledged.

3.2.10 SafeCast revert for specific prices

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.

Recommendation: Consider being very selective when choosing pairs.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.2.11 Oracle <-> Tick deviation

Description: Whenever the tick of a pair deviates from the oracle price, the underlying value of the LP is larger than what is reflected based on fair pricing.

This can result in issues where positions are liquidated earlier than they should because the collateral value is underreported.

Recommendation: Consider implementing a deviation check between TICK <-> ORACLE whenever the price is consulted, based on the fee tier which is decided by governance.

Impermax: We've decided to not implement a deviation check in order to avoid possible downtimes that would bring other kinds of issues.

Auditors: Acknowledged.

3.2.12 Compounding nature in interest accrual

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.

Recommendation: Consider if this issue is desired to be fixed. In that case, the whole interest rate calculation must be refactored.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.2.13 Allowance decrease can be frontran

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.

Recommendation: Consider incorporating `safeIncrease()` and `safeDecrease()` functions.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.2.14 Bad debt created through aggregator update

Description: Since the pricing of LPs in Impermax depends on the price reported by the aggregators of each underlying asset, any non-trivial update made to either aggregator price opens up the opportunity for bad debt to be created in the system.

Consider the following order of events:

- Attacker observes that the market fair price has moved towards the edge of the deviation range OR that the heartbeat duration is almost up
- Attacker creates an LP which is entirely backed by token0
- Attacker borrows 95% token0 and 5% token1 up to their limit of being liquidatable
- In the very next block the position becomes liquidatable
- The attacker liquidates themselves through token0, repaying their entire token0 borrow balance of 95% of their debt
- The attacker is left with a position that is much more exposed to the deviation in price than the original position
- The aggregator update occurs and it causes the position to become underwater
- The attacker restructures the bad debt and liquidates the rest of the position

This way the attacker was able to create bad debt in the market and grief the lenders. It also may be profitable to pull off this attack to extract the liquidation incentive using aggregator updates.

Recommendation: A liquidation limit buffer upon making new borrows would make this attack harder to pull off. Furthermore, using Chainlink aggregators which have a lower deviation threshold will limit the potential magnitude of bad debt which can be created.

Making the restructure bad debt function permissioned would also resolve this issue, though this has been ruled out by the Impermax team.

Impermax: We believe this isn't a risk as long as the safetyMargin is sufficiently larger than the deviation threshold.

Auditors: Confirmed, after a discussion with the team, it was verified that for this scenario to result in bad debt, the position would need to be unhealthy enough that the aggregator update pushes the position from liquidatable to underwater.

3.2.15 Borrowers can avoid bad debt with aggregator updates

Description: When an aggregator update would make a liquidatable position under water, there is an opportunity for bad debt to be avoided by one of the borrowable sides and left for the other borrowable to repay, similar to the

outcome in M-01 `ClaimFees Can Create Bad Debt`.

Consider the following scenario:

- A position is liquidatable and has a borrow balance from both Borrowable A and borrowable B
- An lender in borrowable A sees that an aggregator update is about to occur and frontruns this update to liquidate 1 wei of the position
- The block of last liquidation is now updated to the current block
- The aggregator update occurs in the same block and now the position is under water, the lender from borrowable A now seizes for all of the debt associated with borrowable A so that this debt is fully covered, while also receiving the liquidation bonus.
- The actor then restructures the debt on borrowable B, causing loss for lenders in borrowable B

This way the lender in borrowable A was able to avoid any bad debt while collecting a liquidation fee and griefing those lenders in borrowable B with bad debt.

Recommendation: Consider whether this behavior is undesirable enough to warrant a change in the code.

Impermax: There is no incentive for a lender to wait for the position to become underwater. They would be better off liquidating the position as soon as it becomes liquidatable in order to immediately zero out the risk of bad debt.

Auditors: Confirmed that after a discussion with the team, it was determined the competition dynamic of liquidations should prevent this scenario from happening.

3.3 Low Severity Findings

3.3.1 Use of deprecated chainlink function

Description: The `ImpermaxV3OracleChainlink` contract consults its Chainlink source oracles using the `latestAnswer()` function. According to the Chainlink docs, this function is deprecated. The `latestRoundData()` function can be used instead, and it also returns information such as the timestamp of the latest answer, which can optionally be used to judge if the price is stale (i.e. older than a given threshold). With the specific way the oracle is used within Impermax, adding a revert if the price is stale may cause liveness issues, but given the presence of the `fallbackOracle`, one possible design choice is to use the `fallbackOracle` if the Chainlink source is stale.

Recommendation: To prevent possible future issues with the use of the deprecated `latestAnswer()` function, consider switching to `latestRoundData()` instead. Also, consider adding a staleness check on the latest answer and using the `fallbackOracle` if the price is considered stale.

Impermax: We have decided to not add any staleness check for practical reasons, so using `latestAnswer()` or `latestRoundData().answer` is the same for our scope.

Auditors: Acknowledged.

3.3.2 Contracts callable before initialization

Description: With the `ImpermaxV3Factory` deployment process, the three main contracts for each pool (two borrowable contracts and one collateral contract) are deployed separately via three distinct functions, and a fourth function later calls `_initialize()` on all three. This design raises the question of whether it's possible to interact with borrowable or collateral contracts before they are initialized, and whether this could lead to unexpected behavior.

After analyzing the contracts with this in mind, no major issues were identified. Most functions will revert if the contract is uninitialized, either due to calls to `address(0)` or other logic that fails when key storage variables are not set. Some functions, such as approval-related and admin functions, can be called prior to initialization, but this doesn't seem to introduce any issues.

That said, one edge case is worth mentioning. The permit-based functions (`borrowPermit()` and `permit()`) can be called pre-initialization, at which point the `DOMAIN_SEPARATOR` used for signature verification will be `bytes32(0)` (since it is only set during initialization). So, if an attacker has a victim's signature that was signed using `bytes32(0)` as the domain separator, they could successfully use that signature prior to initialization, and the resulting approval would persist. Fortunately, this seems very unlikely to be exploitable since there isn't any reason an attacker would have such a signature from a victim.

Recommendation: Consider encouraging fully atomic deployments, either by modifying the factory contract or handling it at the frontend level, so that all three contracts are deployed and initialized within a single transaction. Alternatively, since no direct issue was identified with the current deployment flow, consider keeping this behavior in mind and monitoring for any cases where users interact with contracts prior to initialization.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.3 Incorrect rounding for deposits

Description: During the deposit flow the exchange rate is rounded down in the `exchangeRate()` function. This errs on the side of minting more shares to depositors and returning less value to withdawers.

However the system should round against the user in both cases out of an abundance of caution.

Recommendation: Consider using round up division in the `exchangeRate` function only when a deposit is being made and also rounding to minimize the resulting `exchangeRateNew` in the `_mintReserves` function during withdrawals.

Impermax: `exchangeRate()` could be seen as a constant in case the user decides to mint and redeem in the same block. Therefore I would argue that we're already rounding down both in the mint and redeem functions: if a user mints and redeems in the same block his balance will be rounded down twice.

Auditors: Acknowledged.

3.3.4 Separate mul and div vs mulDiv

Description: In several places across the codebase, calculations are performed using a `mul()` then `div()` sequence (multiplying two values and then immediately dividing). In contrast, the UniswapV3 codebase performs these same calculations in a single operation using the FullMath library's `mulDiv()` function, which safely handles overflows in the intermediate multiplication. So, if the `mul()` step overflows a 256-bit value, `mulDiv()` still produces the correct result in UniswapV3, while the exact same calculation in Impermax's code would revert.

In practice, an overflow in the multiplication is unlikely unless extremely large amounts or prices are used, which likely will never happen with normal tokens. However, for consistency and extra safety, using `mulDiv()` would make Impermax's behavior match UniswapV3's behavior.

Recommendation: To match UniswapV3's behavior and prevent possible overflows, consider replacing certain instances of `mul()` then `div()` with `FullMath.mulDiv()`. The relevant contracts that have such calculations include `TokenizedUniswapV3Position`, `CollateralMath`, and `ImpermaxV3OracleChainlink`.

Impermax: Acknowledged. Luckily for most UniV3 math we're already using `mulDiv()` indirectly by using the Uniswap libraries.

Auditors: Acknowledged.

3.3.5 Manager loses out on fees during restructure

Description: In the `ImpermaxV3Borrowable` contract the manager fees are paid out upon calls to mint and withdraw with the underlying `_mintReserves()` function.

The manager fees are a portion of the borrowable token appreciation which comes from interest accrual. The fees are measured based on the amount in excess of the previous `exchangeRate` watermark.

However if no mints or withdrawals occur for an extended period, while yield is accruing through borrowing interest, the manager may lose out on their fees from appreciation when a `restructureDebt` occurs which reduces the exchange rate.

Recommendation: Consider "saving" the manager fees before the reduction of debt occurs in order to checkpoint the appreciation in the vault. The vault manager will then not accrue new fees until the previous exchangeRate is surpassed.

Impermax: This is not a flaw in my opinion. The manager wins when the exchangeRate increases and loses when the exchangeRate decreases. We're not overcharging fees in this way.

Auditors: Acknowledged.

3.3.6 Kink rate decreased before market listing

Description: The `rateUpdateTimestamp` determines how much time has passed since the last borrowing rate update. The borrowing rate adjusts over time with respect to the current utilization, the current kink borrowing rate, and the amount of time which has passed since the last update.

When a borrowable is first deployed through the factory the rateUpdateTimestamp is initialized to the current block time. Therefore the borrowing rate begins to dynamically update right after the createBorrowable0 or createBorrowable1 invocation.

Before a market has been listed on the UI and receives any borrows, the utilization rate will be zero, even if the first deposit has not been made yet. As a result, there is an incentive for an actor to deploy popular markets through the `createNFTLP()` function and to invoke the createBorrowable0 or createBorrowable1 functions. Since there is likely to be a delay between when the NFTLp contract is deployed this way and when the market is listed for deposits and borrows then the borrowing rate will reduce over this period to the minimum of 1% per year over a period of ~2 days.

This could benefit the actor who forced the NFTLp contract to be created and invoked the createBorrowable0 or createBorrowable1 functions prematurely by offering low borrowing rates immediately upon listing.

Recommendation: Consider making the `createNFTLP()` function permissioned so that lending markets are only created when they are intended to be listed.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.7 Zero percentage redeems are allowed

Description: In the redeem function the percentage value is validated to be less than or equal to 1e18, however is not validated to be above zero.

Recommendation: Consider validating that the percentage is above zero to avoid unexpected cases.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.8 Lack of tick range validation

Description: Within the `Tokenization.mint()` function, it is allowed to provide any range for liquidity. While this may be a desired feature, there is very low incentive for legitimate user to create positions with a large range pattern, as they can simply use V2 then.

Therefore, we are of the opinion that it makes sense to limit user flexibility via a settable validation which ensures tick range is in a certain range.

Recommendation: Consider implementing such a validation which allows governance to determine a reasonable range.

Impermax: For certain pairs with high volatility it may still be desirable to provide liquidity in the widest range.

Auditors: Acknowledged.

3.3.9 Nontrivial precision loss for low decimal pairs

Description: In the `getValue()` function the `Q64` value is used to normalize the `amountX.mul(relativePriceX)` and `amountY.mul(relativePriceY)` values. This `Q64` adjustment is applied across the board regardless of the decimals of the underlying tokens being used, in some rare cases where a pair uses low decimal tokens which also have extremely divergent prices this can result in non-trivial precision loss.

For example, if a amountX uses 6 token decimals and the `sqrtPrice` is extremely small, in the range of the tick -500_000 or below, the following precision loss could occur:

- amountX = 1e6
- tick = -600_000
- sqrtPriceX96 = 7425001144658883
- amountX.mul(relativePriceX).div(Q64) = 402.51, whereby 0.51 is truncated off

Precision may start to become notable below tick -500,000 (or above tick 500,000 for the amountY calculation) and becomes significant past tick magnitude 600,000.

Borrowers may be able to leverage this precision loss to borrow amounts which are truncated more aggressively than their collateral, thus allowing them to borrow more than should be safely allowed in such pools.

Recommendation: Consider rounding in the favor of the protocol such that the value of collateral is always rounded down, and the value of debt is always rounded up. Otherwise be sure to refrain from listing any pools where a combination of low decimal tokens and extreme prices can present a rounding issue.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.10 Lacking CEI in claim

Description: In the claim function the collect call is invoked before the feeGrowth is updated for the position stored in the TokenizedUniswapV3Position contract. It is possible for tokens with callbacks to allow untrusted addresses to gain control of the execution flow in the collect function.

Recommendation: No immediate threat has been identified since reentrancy guards are already in place, however out of an abundance of caution consider updating the fee growth of the position prior to making the collect call.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.11 Users cannot directly claim fees

Description: The `claim()` function on the `TokenizedUniswapV3Position` contract is gated by the `_checkAuthorizedCollateral()` validation. This validation assumes that the tokenId is actively being used as collateral in

the `ImpermaxV3Collateral` contract. However for users who are simply holding a `TokenizedUniswapV3Position` tokenId in their EOA directly, they will be unable to claim the fees from the position.

Recommendation: Consider early returning in the `_checkAuthorizedCollateral()` function if the result of `_requireOwned()` is the caller. Additionally, consider checking if the caller is an address which is approved for the tokenId or is approved for all for the owner.

Impermax: I agree that this is a weird behaviour, but NFTLP are designed for the only scope to be used as collateral on Impermax. Considering this, I don't think that fees will need to be claimed in any other scenario.

Auditors: Acknowledged.

3.3.12 Configuration updates retroactively affect fees

Description: The `_setAdjustSpeed()` and `_setKinkUtilizationRate()` functions will affect the value of outstanding borrowing fees which have accrued since the previous update. This may be unexpected for lenders and in some cases may cause some positions to be immediately liquidatable or even under water, though not in an exploitable way since front-running the configuration update would solidify the fees that have accrued.

Recommendation: Add the accrue modifier to the `_setAdjustSpeed()` and `_setKinkUtilizationRate()` functions. Additionally, consider performing `_calculateBorrowRate()` before updating the configurations.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.13 Read only reentrancy risk

Description: In the `liquidate()` function on the borrowable contracts the account debt is not reduced until after the collateral has been seized.

However in the `seize()` function, directly after a portion of the collateral has been seized and directly before the corresponding debt has been forgiven, an ERC721 `safeTransferFrom()` is initiated which gives the liquidator control of the transaction when the position is in a fabricated under water state.

Similarly in the `redeem()` function the caller may gain control over the transaction after the underlying NFTLp has been split and potentially leaves the token in an intermediate under water state.

This could lead to unexpected issues in integrating systems where the intermediate invalid under water state is leveraged.

Recommendation: Be aware of this risk for integrating systems and consider documenting it for anyone composing the Impermax V3 system.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.14 Unexpected unowned tokenId

Description: It is possible to produce an NFTLp collateral tokenId which is unowned by anyone in the `ImpermaxV3Collateral` contract, yet holds a nonzero debt amount. Consider the following scenario:

- Normal setup where tokenId is in the collateral contract and has ample value
- Invoke borrow on borrowable0, get past the validation in `_checkBorrowAllowance()`
- Inside the impermaxV3Borrow callback call redeem with 100% redemption
- provide to as the `ImpermaxV3Collateral` contract
- The borrowBalance checks in redeem pass since the borrow has not completed and been recorded in borrowable0 yet
- Now back in borrow function execution, the borrow is recorded and the `canBorrow()` passes since the `INFTLP(underlying).ownerOf(tokenId)` is indeed the `ImpermaxV3Collateral` contract (but it's not owned by anyone in the `ImpermaxV3Collateral` contract level); and also the collateral is sufficient for the borrow

The resulting state is that a tokenId now exists in the `ImpermaxV3Collateral` contract that is not actually owned by anyone, yet it has nonzero debt. No distinct issues have been identified with this state, however it may pose a risk in the future if updates are made. Furthermore, to avoid any unexpected issues this state should be explicitly prevented.

Recommendation: Consider including the `_requireOwned()` validation in the `canBorrow()` function to prevent this edge case and limit unexpected outcomes.

Impermax: Fixed in commit 068682a.

Auditors: Verified fixed as recommended.

3.3.15 Zero percentage splits allowed

Description: In the split function there is no validation to prevent invocations which would create a new dummy tokenId due to a zero percentage split.

Recommendation: Consider validating that the percentage provided is nonzero to avoid any unexpected edge cases.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.16 100% split can lock unclaimed fees

Description: In `TokenizedUniswapV3Position`, Uniswap's `burn()` function is sometimes called to sync a position with up-to-date fee accumulator values. It should be noted that Uniswap V3 reverts on `burn()` if the position has zero liquidity. This creates an edge case:

- A user creates a position (tokenId1) that is the only tokenId associated with a specific tickLower/tickUpper range, and accumulates fees without claiming them.

- The user decides to do a 100% `split()` into tokenId2.
- tokenId2 is redeemed, removing all liquidity from that specific tick range.
- Any interaction with tokenId1 that calls `burn()` will now revert, making the unclaimed fees inaccessible.

A related mechanism was discussed in the issue "repayToCollateralRatio Allows Unpayable Bad Debt." This issue is meant to reiterate that a manual 100% split can also lead to issues with unclaimable rewards in the original token. Fortunately, this problem can be easily resolved by externally minting 1 wei of liquidity on behalf of the `TokenizedUniswapV3Position` contract.

Recommendation: Consider the consequences of the fact that `TokenizedUniswapV3Position` cannot call Uniswap's `burn()` function when it owns 0 liquidity in the given tick range. If the suggested fix above (manually adding 1 wei of liquidity for any "stuck" tokenIds) is acceptable in the worst case, consider simply disallowing 100% splits in the frontend so this situation is less likely to occur.

Impermax: Acknowledged. It's good that in the collateral `redeem()` function we don't allow users to do a 100% split. Technically this could happen if `seizePercentage` equals 100% inside the `seize()` function, but it's a very rare scenario.

Auditors: Acknowledged.

3.3.17 Possible loss of fee accumulation

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.

Recommendation: Consider ensuring that mint is always called in the same transaction as the liquidity is added.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.18 Multiple fee tiers increase flexibility

Description: Currently it is possible to push any pool fee tier to be used in the protocol. While we could not identify any immediate threat from this logic, we are the opinion that governance should decide which pools are allowed, as this does not affect user experience.

Recommendation: Consider only allowing governance to push new pools.

Impermax: Acknowledged. We'd rather keep this fully permissionless

Auditors: Acknowledged.

3.3.19 Zero redemption possibility

Description: Within the ImpermaxV3Collateral contract, it is possible to call redeem with `percentage = 0`. This is an invalid operation and should be properly validated.

Recommendation: Consider ensuring that `percentage != 0`.

Impermax: Acknowledged. We skip a lot of checks for "wasteful but harmless" actions (for instance redeeming 0, splitting 0, etc). For consistency we're going to skip also this one.

Auditors: Acknowledged.

3.3.20 Unpaid interest distribution

Description: As with many lending protocols, ImpermaxV3 distributes interest which has not been paid back. While this is not an issue itself, it can be leveraged in different scenarios.

Recommendation: Consider acknowledging this issue.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.21 Validation of reasonable price

Description: As already mentioned in another issue, one important validation is to ensure reasonable tick ranges. The same applies for extreme prices.

Recommendation: Consider implementing a maximum upper/lower tick. This can be settable by governance based on the pair and situation

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.22 Lack of explicitness in borrow

Description: The `borrow()` function handles repayments and borrowing. This is simply determined on the input parameter and the balance change. There is currently no explicit enforcement that this function treats only one of both actions. Therefore, it is possible to execute both actions at the same time, which is pointless from a user perspective.

It is for example possible to provide a `repayAmount` while at the same time borrowing funds from the contract. While this is indeed properly handled within this function and within the `_updateBorrow()` function, it is worth a consideration to make the borrow function more strict, such as forcing `borrowAmount` to be zero in a repayment scenario and vice versa. It may be possible that this can lead to unexpected behavior.

Recommendation: Consider thinking if it makes sense to make the `borrow()` function more strict.

Impermax: The reason why this is kept open is to allow for flashloans. There are also other practical scenarios where you may want to borrow and repay within the same operation. For instance if you want to do a swap getting a certain `amountOut` without initially knowing the `amountIn`. You could initially borrow a high amount of tokens, then use what's necessary to do the swap, and finally send the rest back to the contract. A variation of this loan/flashloan approach was used during the leverage operation.

Auditors: Acknowledged.

3.3.23 Lack of closeFactor usage

Description: Most lending protocols incorporate a so-called closeFactor. This determines how much of an outstanding debt can be repaid and is responsible for protecting borrowers to not be fully liquidated once their positions become unhealthy.

Within ImpermaxV3, there is no closeFactor which essentially allows positions to be liquidated fully, resulting in serious disadvantage for borrowers.

Recommendation: Consider acknowledging this issue, as implementing a closeFactor would be a major code change at this point.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.24 Unused callback within redeem

Description: Within the redeem function, a callback is introduced which essentially calls back to the msg.sender. Currently we could not figure out a legitimate usecase for this callback (unlike during liquidation flow). Therefore, it might make sense to remove this callback.

Recommendation: Consider clarifying the usecase for the callback within redeem or simply removing it. It has however to be noted that the `safeTransferFrom()` itself also offers callback logic.

Impermax: The impermaxV3Redeem callback is used during the deleverage operation. For instance, a user can redeem 100% of his collateral, remove the liquidity, and finally use the funds to repay his debt. In this way he can close his position in an efficient way.

Auditors: Acknowledged.

3.3.25 borrowBalance function is unupdated

Description: Currently, the `borrowBalance()` function does not incorporate any interest since the last update. This can result in unexpected side-effects for protocols that integrate on top of ImpermaxV3. This is already indicated in the NATSPEC: "// this is the stored borrow balance; the current borrow balance may be slightly higher"

Recommendation: Consider extending NATSPEC by adding a comment that the contract state should be updated before consulting `borrowBalance()`.

Impermax: Acknowledged.

Auditors: Acknowledged.

3.3.26 Lack of basic return value checks

Description: The oraclePriceSqrtX96 function in the ImpermaxV3OracleChainlink contract exhibits several integration issues that could lead to potential vulnerabilities: 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. 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. 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. 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.

Recommendation: Consider whether it makes sense to implement these mentioned safeguards. However, generally speaking, Chainlink oracles are usually deemed as safe.

Impermax: The reason why we opted for this design was because of the lack of a reliable alternative in case the main feed was down. Reverting a transaction after any of these check would make the oracle unreliable and could lock user's funds for an extended period of time.

Auditors: Acknowledged.