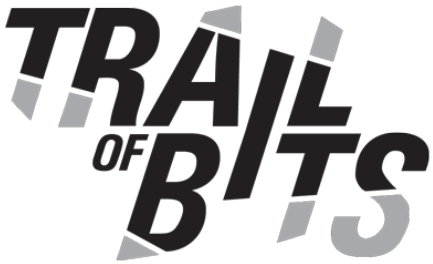

# Uniswap V3 Core

## Security Assessment

**March 12, 2021**


Prepared For:
Hayden Adams | *Uniswap*
hayden@uniswap.org

Prepared By:
Alexander Remie | *Trail of Bits*
alexander.remie@trailofbits.com

Dominik Teiml | *Trail of Bits*
dominik.teiml@trailofbits.com

Josselin Feist | *Trail of Bits*
josselin.feist@trailofbits.com

# Executive Summary

During the week of January 4, 2021 and from February 15 to March 12, 2021, Uniswap engaged Trail of Bits to review the security of Uniswap V3. Trail of Bits conducted this assessment over 10 person-weeks, with 3 engineers working from 99223f3 from the uniswap-v3-core repository.

In the first week (in January), we focused on gaining a high-level understanding of the project. We started by reviewing the three main contracts against the most common Solidity flaws and found the first two issues. Because the second week of our engagement was several weeks later, we first reviewed the diff of the code since the first week. We then studied the updated whitepaper and reviewed the factory contract and the mint/burn and flash functionalities.

In the third week, Trail of Bits focused on the math libraries and the swap function. In the fourth, we continued our manual review of the arithmetic libraries, the flash loan feature, and pool initialization and focused on using Echidna to test properties. In the final week, we added more Echidna properties to the core pool contracts and libraries and improved the existing properties by adding dynamic position creation. This enabled us to discover issues such as TOB-UNI-010.

We found 10 issues, including 2 of high severity. The most critical is TOB-UNI-005, which allows anyone to drain a pool's funds in both tokens due to an incorrect balance comparison.

Uniswap developed a significant set of properties and used Echidna to ensure the correctness of the arithmetic, including rounding. The system includes one of the broader sets of properties in the industry and demonstrates Uniswap's commitment to ensuring the security of its protocol.

Overall, the codebase follows best practices. The code is well structured, and Uniswap avoided the most common Solidity pitfalls. However, due to the novelty of the project, it suffers from significant complexity. The state of the whitepaper, a work in progress, made the code review more difficult and increased the likelihood of issues. Additionally, drastic gas optimizations such as a lack of SafeMath and the assembly usage increase the probability of undiscovered bugs. While there is significant testing on the individual components, the system will benefit from more thorough end-to-end tests on the overall swapping, minting, and burning process.

Trail of Bits recommends that Uniswap complete the following:

- Address all reported issues.

- Expand the documentation of arithmetic functions with precise assumptions about the ranges of all inputs and outputs.
- Add unit tests and Echidna tests for libraries and core contracts, particularly the `LiquidityMath`, `Tick`, and `Position` libraries and the pool and factory contracts.
- Improve unit test and Echidna test coverage for the end-to-end system properties.
- Consolidate and finish the whitepaper.
- Conduct a security review of the periphery contracts, focusing on ensuring that their interactions with the core match the core's assumptions.

# Project Dashboard

**Application Summary**

| | |
|---|---|
| Name | Uniswap V3 Core |
| Version | 99223f3 |
| Type | Solidity |
| Platforms | Ethereum |

**Engagement Summary**

| | |
|---|---|
| Dates | Week of January 4, 2021 and February 15 – March 12, 2021 |
| Method | Whitebox |
| Consultants Engaged | 3 |
| Level of Effort | 10 person-weeks |

**Vulnerability Summary**

| | | |
|---|---|---|
| Total High-Severity Issues | 2 | ■ ■ |
| Total Medium-Severity Issues | 4 | ■ ■ ■ ■ |
| Total Low-Severity Issues | 1 | ■ |
| Total Informational-Severity Issues | 3 | ■ ■ ■ |
| Total | 10 | |

**Category Breakdown**

| | | |
|---|---|---|
| Data Validation | 6 | ■ ■ ■ ■ ■ ■ |
| Undefined Behavior | 2 | ■ ■ |
| Timing | 1 | ■ |
| Auditing and Logging | 1 | ■ |
| Total | 10 | |

## Code Maturity Evaluation

| Category Name | Description |
|---|---|
| Access Controls | **Satisfactory**. The number of public-facing functions is limited, and the access controls are satisfactory. However, one issue related to access controls (TOB-UNI-001) was found, and the system would benefit from clear documentation on the owner's privileges. |
| Arithmetic | **Moderate**. Overall, Uniswap has devoted significant effort to making arithmetic operations (including custom ones) safe. However, we identified several such issues (TOB-UNI-005 and TOB-UNI-010), and Uniswap identified additional issues during the review. The arithmetic would also benefit from more robust edge cases and more thorough testing on the end-to-end operations. |
| Assembly Use | **Satisfactory**. Assembly is used extensively in two complex, critical functions, `mulDiv` and `getTickAtSqrtRatio`. Writing these functions in Solidity would decrease risks to the system. |
| Centralization | **Satisfactory.** The system is parameterized by the factory owner. The owner can add new available (`fee`, `tickSpacing`) pairs in the factory, depending on data validation. In the pool, the owner can collect protocol fees and include them among a set of available options. In general, the owner does not have unreasonable power. However, the system would benefit from more restrictions on the system parameters' values (see TOB-UNI-006). |
| Upgradeability | **Not applicable.** The system cannot be upgraded. |
| Function Composition | Satisfactory. Overall, the code is well structured. Most logic is located in one of the numerous libraries, and logic is extracted into pure functions whenever possible. The splitting of the code into logical libraries is a good practice and makes unit testing and fuzzing the system much easier. However, the system would benefit from schema describing the different components and their interactions and behaviors. |
| Front-Running | **Satisfactory**. We did not find many issues regarding front-running. In the mint and burn functionality, we did not see a way for a front-runner to profit. A front-runner may generate profits from the swap functionality, as in V1 and V2, but the loss incurred by the user is mitigated by the limit price. Finally, the initialization of pools can be front-run (TOB-UNI-007). Due to its nature, the system allows for arbitrage opportunities; documentation regarding those opportunities would be beneficial to users. |

| | |
|---|---|
| Monitoring | **Satisfactory.** In general, functions emit events where appropriate. However, in at least one case, validation is not performed, which can cause such an emission to be misleading (TOB-UNI-002). |
| Specification | **Moderate.** At the beginning of the assessment, the whitepaper provided by Uniswap was not up to date with the codebase. Many sections were missing, and it underwent significant changes during the review. Appendix D contains our initial recommendations. While the specification has improved, it is still a work in progress, making the code review more difficult. |
| Testing & Verification | **Moderate.** The project has extensive but incomplete unit tests and Echidna tests. Uniswap devoted significant effort to testing the individual components, but the tests lack end-to-end coverage. More thorough end-to-end coverage would have discovered issue TOB-UNI-005, which allows anyone to drain a pool. |

## Engagement Goals

The engagement was scoped to provide a security assessment of the Uniswap V3 smart contracts.

Specifically, we sought to answer the following questions:

- Are all arithmetic libraries correct?
- Are the arithmetic libraries used correctly, and do they correctly apply rounding?
- Do the main interactions with the contracts lead to expected behavior?
- Are there appropriate access controls for privileged actions?
- Is it possible to manipulate the price and gain an unfair advantage when executing swaps?
- Are the pool operations susceptible to front-running issues?
- Is it possible to perform swaps without paying the required amount?
- Is it possible to drain funds from a pool?

## Coverage

**Arithmetic primitives (BitMath, FullMath, UnsafeMath, SafeCast, and LowGasSafeMath).** These libraries form the mathematical building blocks of the system. For most functions, we extensively reviewed the implementations to ensure that they would return the correct results and revert otherwise. For example, `safeAdd` should return the sum if the mathematical sum is at most 2^256 - 1 and should revert if it is not. Safe casts should return a new type if the old value fits and should otherwise revert. For all functions, we completed a comprehensive review of the Echidna tests, checking that their properties sufficiently modeled the desired behavior. We also reviewed the unit tests for all functions and again confirmed that the returned values were sufficiently constrained.

**TransferHelper.** TransferHelper contains just one function, safeTransfer. We manually checked how the lack of a contract existence check would affect the operations of the pool contract that used safeTransfer. We also checked that the possible transfer return values were all correctly handled.

**LiquidityMath.** LiquidityMath contains just one function, addDelta. We checked for both underflow and overflow cases and verified that the correct result was returned in the success case.

**TickMath.** TickMath defines four constants and two functions used to convert prices to ticks. Both functions have very complex implementations; one makes extensive use of assembly for gas optimization. We manually checked that the conversion of the input argument was correct. We also ran extensive tests, using the existing Echidna tests, to discern whether we could trigger an out-of-bounds return value.

**TickBitmap.** TickBitmap defines three functions to obtain the position of a tick, flip a tick, and retrieve the next initialized tick. We manually reviewed the conversion of ticks to the wordPos and bitPos. We also reviewed the use of bitwise XOR to update the bitmap. For the last function, we ran extensive tests, using the existing Echidna tests, to check whether we could trigger an assertion.

**Tick.** This defines the tick struct, one pure function to calculate maximum liquidity per tick, and four methods that operate on a tick struct. We studied the method by which tickSpacing is enforced on the MIN_TICK and MAX_TICK and the corresponding code that determines the number of ticks. We also examined the function to cross a tick to see if it correctly updated all tick struct variables.

**Position.** This defines the position struct, a getter function that operates on the position mapping, and an update function employed when a user wants to add or remove liquidity. We examined the process of creating a position key to see if it was possible to create overlapping position keys. We performed a manual review and wrote several unit tests to determine if the update function correctly calculated and updated the fee-related position struct variables, including in cases in which liquidityDelta was below zero, equal to zero, and above zero.

**SqrtPriceMath.** This library contains formulas that operate between the prices, liquidities, and token amounts. These functions are used

- to identify the amount of tokens a user must transfer/receive in order to add/remove liquidity from a position,
- to identify what sizes of orders can be fulfilled based on the limit price and the price of the next initialized tick within one word, and
- to identify the next price if an order has been partially filled.

We were able to manually check the implementations of the `getAmount(0|1)Delta` and the `getNextSqrtPriceFrom(Input|Output)` functions but not the `getNextSqrtPriceFromAmount(0RoundUp|1RoundingDown)` functions. We checked for correct signs, over- and underflow, and the correct handling of rounding.

**SwapMath.** This library contains just one function, `computeSwapStep`, which computes the size of an order to fulfill, based on the current step parameters. Since this function fulfills all four cases of (exactIn, zeroForOne) possibilities, we checked whether the implementation was correct for all situations. We also checked that the correct type of rounding (i.e., up or down) was used in all situations.

**Oracle.** This library contains a struct and provides functions to store (historical) liquidity and the tick values of the pool's tokens. We briefly reviewed the implementation, checking that updating the cardinality preserved the monotonicity of the observations.

**SecondsOutside.** This library is used to track the amount of time that a position has been active. It is gas-optimized by packing eight 32-bit timestamps inside one uint256. We reviewed the use of bitwise operations to determine the bit-shift and wordPos of a tick. We also confirmed that bitwise operations in the cross and clear functions were correct.

**NoDelegateCall.** This contract prevents the execution of delegate calls into Uniswap contracts and is applied to most of the functions in the factory and pool contracts. We manually checked that the implementation prevented delegate calls and did not lead to any unexpected results.

**UniswapV3Factory.** This contract contains methods that allow anyone to create a new pool, as well as owner-only methods that enable the creation of a new (`fee`, `tickSpacing`) pair and owner changes. We checked that a new fee and tickSpacing could not overwrite existing entries. We also checked that the creation of a new pool correctly shuffled the tokens when necessary and that no pool could overwrite an existing pool. In addition, we inspected the chosen range of permitted tickSpacing values and its effect on the pool contract. Lastly, we checked how control of the ownership role could be lost during deployment or reassignment of the owner role.

**UniswapV3PoolDeployer.** This contract contains just one function to deploy a new pool contract using CREATE2. We checked that the chosen arguments for the CREATE2 address did not lead to any problems. We also checked that the use of CREATE2 would not cause any issues (e.g., that it would not cause a pool to self-destruct).

**UniswapV3Pool.** This is the core contract of the Uniswap V3 project. Its main functions are mint, burn, swap, and flash. Its numerous other functions include functions that update protocol fee percentages, withdraw protocol fees, and withdraw a position's collected fees, as well as an initialize function to set an initial price upon deployment and various view functions. For this contract, we performed an extensive manual review, wrote unit tests, and wrote end-to-end Echidna tests.

We checked if the price could be manipulated through swap, mint, burn, or initialize and if an attacker could manipulate the price to swap tokens at an unfair price. We also examined how swaps of various amounts could be used to move the current price and if the price movement could be exploited by an attacker to swap tokens at an unfair price. Additionally, we reviewed the process of front-running an initialize call and how **such a call** could be used by an attacker to execute (and profit off of) a swap at an unfair price. We assessed the flash loan function to see if an attacker could use it without repaying the loan (+fee), as well as the loop inside the swap function to see whether it could cause a denial of service due to a large amount of very small ticks. We also inspected the swap, burn, and mint callback functions, with an eye toward confirming that the surrounding checks were correctly implemented to prevent minting/swapping at no cost. We examined various non-standard

ERC20 tokens and how they could lead to unexpected results when used in a pool, confirmed that all functions inside the pool contract were implemented correctly, and checked the validation of all function parameters. Lastly, we implemented more than 25 Echidna end-to-end properties to test various invariants for mint, burn, and swap.

Due to time constraints, Trail of Bits could not explore the following areas:

- The Oracle functions, with the exception of those that deal with increasing the cardinality
- The last three lines of the `computeSwapStep` function, which deal with determining the `feeAmount`
- `TickBitmap.nextInitializedTickWithinOneWord`
- `SecondsOutside.secondsInside`
- Assembly inside `TickMath.getTickAtSqrtRatio`

# Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance the coverage of certain areas of the contracts, including the following:

- Slither, a Solidity static analysis framework.
- Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation.

Automated testing techniques augment our manual security review but do not replace it. Each method has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode, and Echidna may not randomly generate an edge case that violates a property.

## Automated Testing with Echidna

We wrote more than 25 end-to-end properties. Because Uniswap had already implemented many per-component Echidna tests, we decided to set up an end-to-end Echidna test suite. We wrote Echidna tests for the swap, mint, and burn functions and achieved sufficient Echidna test coverage throughout those functions. We implemented the following Echidna properties:

### End-to-End Properties

| ID | Property | Result |
|---|---|---|
| 1 | Calling `mint` never leads to a decrease in `liquidity`. | PASSED |
| 2 | Calling `mint` always leads to an increase in `ticks(tickLower).liquidityGross`. | PASSED |
| 3 | Calling `mint` always leads to an increase in `ticks(tickUpper).liquidityGross`. | PASSED |
| 4 | Calling `mint` always leads to an increase in `ticks(tickLower).liquidityNet`. | PASSED |
| 5 | Calling `mint` always leads to a decrease in `ticks(tickUpper).liquidityNet`. | PASSED |
| 6 | Calling `mint` always reverts if neither `tickLower` nor `tickUpper` is a multiple of the configured `tickSpacing`. | PASSED |
| 7 | Calling `burn` never leads to an increase in `liquidity`. | PASSED |

| | | |
|---|---|---|
| 8 | Calling `burn` does not lead to an increase in `ticks(tickLower).liquidityGross`. | PASSED |
| 9 | Calling `burn` does not lead to an increase in `ticks(tickUpper).liquidityGross`. | PASSED |
| 10 | Calling `burn` does not lead to an increase in `ticks(tickLower).liquidityNet`. | PASSED |
| 11 | Calling `burn` does not lead to a decrease in `ticks(tickUpper).liquidityNet`. | PASSED |
| 12 | Calling `burn` always reverts if neither `tickLower` nor `tickUpper` is a multiple of the configured `tickSpacing`. | PASSED |
| 13 | Calling `swap` with `zeroForOne` does not lead to a decrease in `feeGrowthGobal0X128`. | PASSED |
| 14 | Calling `swap` with `zeroForOne` does not lead to a change in `feeGrowthGobal1X128`. | PASSED |
| 15 | Calling `swap` with `!zeroForOne` does not lead to a decrease in `feeGrowthGobal1X128`. | PASSED |
| 16 | Calling `swap` with `!zeroForOne` does not lead to a change in `feeGrowthGobal0X128`. | PASSED |
| 17 | If calling `swap` does not change the `sqrtPriceX96`, `liquidity` will not change. | PASSED |
| 18 | If calling `swap` with `zeroForOne` does not lead to the payment of token0, it will not lead to the receipt of `token1`. | PASSED |
| 19 | If calling `swap` with `!zeroForOne` does not lead to the payment of `token1`, it will not lead to the receipt of `token0`. | PASSED |
| 20 | `liquidityNet` over all ticks should sum to zero. | PASSED |
| 21 | `liquidity` is equal to the summation of `liquidityNet` for all ticks below and including the current `tick`. | PASSED |
| 22 | For the ticks immediately below (`t_b`) and above (`t_a`) the current tick, `ticks[t_b].feeGrowthOutside0X128 + ticks[t_a].feeGrowthOutside0X128 <= feeGrowthGlobal0X128`. | PASSED |
| 23 | For the ticks immediately below (`t_b`) and above (`t_a`) the current tick, `ticks[t_b].feeGrowthOutside1X128 + ticks[t_a].feeGrowthOutside1X128 <= feeGrowthGlobal1X128`. | PASSED |

| | | |
|---|---|---|
| 24 | `feeGrowthGlobal0X128` and `feeGrowthGlobal1X128` are non-strictly increasing in calls to swap. | PASSED |
| 25 | After a `mint`, calling the inverse `burn` always succeeds. | PASSED |
| 26 | Calling `burn` on an existing position with amount zero never fails. | PASSED |
| 27 | Burning x amount of a position always decreases `position.liquidity` by x amount. | PASSED |
| 28 | Burning less than the total position amount never fails. | PASSED |
| 29 | Calling `burn` with amount zero does not change the `liquidity` of the pool. | PASSED |

### Arithmetic Properties

| ID | Property | Result |
|---|---|---|
| 30 | `getNextSqrtPriceFromInput/getNextSqrtPriceFromOutput` always returns a price between MIN_SQRT_RATIO and MAX_SQRT_RATIO (inclusive). | **FAILED (TOB-UNI-010)** |

## Verification with Manticore

Verification was performed with the experimental branch `dev-evm-experiments`, which contains new optimizations and is a work in progress. Trail of Bits will ensure that the following properties hold once the branch has stabilized and been included in a Manticore release:

| ID | Property | Result |
|---|---|---|
| 1 | `BitMath.mostSignificantBit` returns a value in `x >= 2**msb && (msb == 255 || x < 2**(msb+1))`. | VERIFIED |
| 2 | `BitMath.leastSignificantBit` returns a value in `((x & 2** lsb) != 0) && ((x & (2**(lsb -1))) == 0)`. | VERIFIED |
| 3 | If `LiquidityMath.addDelta` returns, the value will be equal to `x + uint128(y)`. | VERIFIED |

## Manual Verification

| ID | Property | Result |
|---|---|---|
| 1 | `amountIn + feeAmount` cannot overflow | VERIFIED (Appendix H) |

## Automated Testing with Slither

We implemented the following Slither property:

| Property | Result |
|---|---|
| Every publicly accessible function uses the `lock` modifier, is whitelisted, or is a `view` function. | PASSED (APPENDIX F) |

# Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

## Short Term

❑ **Designate msg.sender as the initial owner of a pool contract, and implement a two-step ownership-transfer process.** This will ensure that the owner role is not assigned to an address not controlled by any user. TOB-UNI-001

❑ **Add a check ensuring that the `_owner` argument does not equal the existing `owner`.** This check will prevent the emission of an event indicating that the owner role was changed when it was actually reassigned to the current owner. TOB-UNI-002

❑ **Add documentation explaining to users that the use of interest-earning tokens can reduce the standard payments for minting and flash loans.** That way, they will not be surprised if they use an interest-earning token through Uniswap. TOB-UNI-003

❑ **Correct the sentence in the whitepaper regarding the effect of price movements on the number of tokens that are touched.** This will prevent the whitepaper's readers from becoming confused. TOB-UNI-004

❑ **Replace the `>=` with `<=` inside the `require` in the `swap` function and add at least one unit test checking that the `IIA` error is thrown when too few tokens are transferred from the initiator's contract to the pool.** The current logic allows an attacker to drain the pool. TOB-UNI-005

❑ **Determine a reasonable minimum tick spacing requirement, or consider setting a minimum for liquidity per position.** This will lower the likelihood of a DoS in the while loop. TOB-UNI-006

❑ **Consider moving price initialization operations to the constructor, adding access controls to the initialize function, or enhancing the documentation to warn users against price manipulation through the initialize function.** This will lower the risk of users unknowingly falling victim to price manipulation during initialization of the pool. TOB-UNI-007

❑ **There does not appear to be a straightforward way to prevent TOB-UNI-008. We recommend investigating the limits associated with pools without liquidity in some**

**ticks and ensuring that users are aware of the risks so that they can make informed decisions.** TOB-UNI-008

❑ **Check the contract's existence prior to the low-level call in `TransferHelper.safeTransfer`.** This will ensure that a swap reverts if the token to be bought no longer exists, preventing the pool from accepting the token to be sold without returning any tokens in exchange. TOB-UNI-009

❑ **Check in `getNextSqrtPriceFromInput/getNextSqrtPriceFromOutput` that the returned value is within `MIN_SQRT_RATIO`, `MAX_SQRT_RATIO`.** Including the check where the calculation is performed will reduce the likelihood that a refactor will remove the check and cause problems in calling functions. TOB-UNI-010

## Long Term

❑ **Use Slither, which will catch the missing `address(0)` check.** Using Slither will also prevent important privileged roles from being assigned to address zero, which causes a permanent loss of access to the role. TOB-UNI-001

❑ **Carefully inspect the code to ensure that configuration functions do not allow a value to be updated as the existing value.** This will prevent the emission of an event falsely indicating a change. TOB-UNI-002

❑ **Using the relevant recommendations in the token integration checklist (Appendix E), generate a document detailing the shortcomings of tokens with certain features and the impacts of their use in the Uniswap V3 protocol.** That way, users will not be alarmed if the use of a token with non-standard features leads to unexpected results. TOB-UNI-003

❑ **Finalize the whitepaper, ensuring that it is clear.** Enable as many users as possible to read and understand the whitepaper and the inner workings of Uniswap. TOB-UNI-004

❑ **Consider adding more properties and using Echidna or Manticore to verify that initiators are correctly transferring tokens to the pool.** The current tests did not catch a critical bug in the swap callback. TOB-UNI-005

❑ **Consider adding at least one unit test for each error that can be thrown by the contracts.** With a unit test, each error in the contract/libraries would be thrown when it should be, at least in simple cases. TOB-UNI-005

❑ **Make sure that all parameters that the owner can enable (such as fee level and tick spacing) have bounds that lead to expected behavior, and clearly document those bounds, such as in a markdown file or in the whitepaper.** Documentation would allow users to inspect the enabled fee levels and tick spacings, which could affect whether they decide to use a specific pool or to create one with the desired fee and tick spacing. TOB-UNI-006

❑ **Avoid initialization outside of the constructor. If that is not possible, ensure that the underlying risks of initialization are documented and properly tested.** Initialization done outside of the constructor is error-prone and a bad practice and can lead to contract compromise. TOB-UNI-007

❑ **Ensure that pools can never end up in an unexpected state.** This will ensure that the system's behavior is predictable at all states. TOB-UNI-008

❑ **Avoid low-level calls. If such a call is not avoidable, carefully review the Solidity documentation, particularly the "Warnings" section.** This will protect against unforeseen (missing) features of the Solidity language. TOB-UNI-009

❑ **Document every bound for all arithmetic functions and test every bound with Echidna and Manticore.** Documentation will ensure that each function's bounds are immediately clear, and testing will ensure that functions do not return out-of-bound values. TOB-UNI-010

# Findings Summary

| # | Title | Type | Severity |
|---|---|---|---|
| 1 | Missing validation of `_owner` argument could indefinitely lock owner role | Data Validation | Medium |
| 2 | Missing validation of `_owner` argument could lead to incorrect event emission | Auditing and Logging | Informational |
| 3 | Anyone could steal pool tokens' earned interest | Timing | Low |
| 4 | Whitepaper contains incorrect equation | Undefined Behavior | Informational |
| 5 | Incorrect comparison enables swapping and token draining at no cost | Undefined Behavior | High |
| 6 | Unbound loop enables denial of service | Data Validation | Medium |
| 7 | Front-running pool's initialization can lead to draining of liquidity provider's initial deposits | Data Validation | Medium |
| 8 | Swapping on zero liquidity allows for control of the pool's price | Data Validation | Medium |
| 9 | Failed transfer may be overlooked due to lack of contract existence check | Data Validation | High |
| 10 | `getNextSqrtPriceFromInput\|Output` can return a value outside of `MIN_SQRT_RATIO, MAX_SQRT_RATIO` | Data Validation | Informational |

## 1. Missing validation of _owner argument could indefinitely lock owner role

Severity: Medium
Type: Data Validation
Target: `UniswapV3Factory.sol`

Difficulty: High
Finding ID: TOB-UNI-001

**Description**
A lack of input validation of the `_owner` argument in both the `constructor` and `setOwner` functions could permanently lock the owner role, requiring a costly redeploy.

```
    constructor(address _owner) {
        owner = _owner;
        emit OwnerChanged(address(0), _owner);

        _enableFeeAmount(600, 12);
        _enableFeeAmount(3000, 60);
        _enableFeeAmount(9000, 180);
    }
```

*Figure 1.1:* `constructor` *in* `UniswapV3Factory.sol`.

```
    function setOwner(address _owner) external override {
        require(msg.sender == owner, 'OO');
        emit OwnerChanged(owner, _owner);
        owner = _owner;
    }
```

*Figure 1.2:* `setOwner` *in* `UniswapV3Factory.sol`.

The `constructor` calls `_enableFeeAmount` to add three available initial fees and tick spacings. This means that, as far as a regular user is concerned, the contract will work, allowing the creation of pairs and all functionality needed to start trading. In other words, the incorrect owner role may not be noticed before the contract is put into use.

The following functions are callable only by the owner:

- `UniswapV3Factory.enableFeeAmount`
  - Called to add more fees with specific tick spacing.
- `UniswapV3Pair.setFeeTo`
  - Called to update the fees' destination address.
- `UniswapV3Pair.recover`
  - Called to withdraw accidentally sent tokens from the pair.
- `UniswapV3Factory.setOwner`
  - Called to change the owner.

To resolve an incorrect owner issue, Uniswap would need to redeploy the factory contract and re-add pairs and liquidity. Users might not be happy to learn of these actions, which

could lead to reputational damage. Certain users could also decide to continue using the original factory and pair contracts, in which owner functions cannot be called. This could lead to the concurrent use of two versions of Uniswap, one with the original factory contract and no valid owner and another in which the owner was set correctly.

Trail of Bits identified four distinct cases in which an incorrect owner is set:
- Passing `address(0)` to the `constructor`
- Passing `address(0)` to the `setOwner` function
- Passing an incorrect address to the `constructor`
- Passing an incorrect address to the `setOwner` function.

**Exploit Scenario**
Alice deploys the `UniswapV3Factory` contract but mistakenly passes `address(0)` as the `_owner`.

**Recommendation**
Several improvements could prevent the four abovementioned cases:
- Designate `msg.sender` as the initial owner, and transfer ownership to the chosen owner after deployment.
- Implement a two-step ownership-change process through which the new owner needs to accept ownership.
- If it needs to be possible to set the owner to `address(0)`, implement a `renounceOwnership` function.

Long term, use Slither, which will catch the missing `address(0)` check, and consider using two-step processes to change important privileged roles.

## 2. Missing validation of `_owner` argument could lead to incorrect event emission

Severity: Informational
Type: Auditing and Logging
Target: `UniswapV3Factory.sol`

Difficulty: High
Finding ID: TOB-UNI-002

**Description**
Because the `setOwner` lacks input validation, the owner can be updated to the existing owner. Although such an update wouldn't change the contract state, it would emit an event falsely indicating the owner had been changed.

```
    function setOwner(address _owner) external override {
        require(msg.sender == owner, 'OO');
        emit OwnerChanged(owner, _owner);
        owner = _owner;
    }
```

*Figure 2.1:* `setOwner` *in* `UniswapV3Factory.sol`.

**Exploit Scenario**
Alice has set up monitoring of the `OwnerChanged` event to track transfers of the owner role. Bob, the current owner, calls `setOwner` to update the owner to his address (not actually making a change). Alice is notified that the owner was changed but upon closer inspection discovers it was not.

**Recommendation**
Short term, add a check ensuring that the `_owner` argument does not equal the existing `owner`.

Long term, carefully inspect the code to ensure that configuration functions do not allow a value to be updated as the existing value. Such updates are not inherently problematic but could cause confusion among users monitoring the events.

## 3. Anyone could steal pool tokens' earned interest

Severity: Low
Type: Timing
Target: `UniswapV3Pool.sol`

Difficulty: Medium
Finding ID: TOB-UNI-003

**Description**
Unexpected ERC20 token interest behavior might allow token interest to count toward the amount of tokens required for the `UniswapV3Pool.mint` and `flash` functions, enabling the user to avoid paying in full.

The mint function allows an account to increase its liquidity in a position. To verify that the pool has received at least the minimum amount of tokens necessary, the following code is used:

```
    uint256 balance0Before;
    uint256 balance1Before;
    if (amount0 > 0) balance0Before = balance0();
    if (amount1 > 0) balance1Before = balance1();
    IUniswapV3MintCallback(msg.sender).uniswapV3MintCallback(amount0, amount1, data);
    if (amount0 > 0) require(balance0Before.add(amount0) <= balance0(), 'M0');
    if (amount1 > 0) require(balance1Before.add(amount1) <= balance1(), 'M1');
```

*Figure 3.1: `UniswapV3Pool.sol#L384-L390`*

Assume that both `amount0` and `amount1` are positive. First, the current balances of the tokens are fetched. This step is followed by a call to the `uniswapV3MintCallback` function of the caller, which should transfer the required amount of each token to the pool contract. Finally, the code verifies that each token's balance has increased by at least the required amount.

A token could allow token holders to earn interest simply because they are token holders. It is possible that to retrieve this interest, any token holder could call a function to calculate the interest earned and increase the token holder's balance.

An attacker could call the function to pay out interest to the pool contract from within the `uniswapV3MintCallback` function. This would increase the pool's token balance, decreasing the number of tokens that the user needs to transfer to the pool contract in order to pass the balance check (i.e., the check confirming that the balance has sufficiently increased). In effect, the user's token payment obligation is reduced because the interest accounts for part of the required balance increase.

To date, we have not identified a token contract that contains such a functionality; however, it is possible that one could exist or be created.

Similarly, the `flash` function allows any user to secure a flash loan from the pool.

**Exploit Scenario**
Bob deploys a pool with token1 and token2. Token1 allows all of its holders to earn passive interest. Anyone can call `get_interest(address)` to make a specific token holder's interest be claimed and added to the token holder's balance. Over time, the pool can claim 1,000 tokens. Eve calls `mint` on the pool, such that the pool requires Eve to send 1,000 tokens. Eve calls `get_interest(address)` instead of sending the tokens, adding liquidity to the pool without paying.

**Recommendation**
Short term, add documentation explaining to users that the use of interest-earning tokens can reduce the standard payments for minting and flash loans.

Long term, using the token integration checklist (Appendix E), generate a document detailing the shortcomings of tokens with certain features and the impacts of their use in the Uniswap V3 protocol. That way, users will not be alarmed if the use of a token with non-standard features leads to unexpected results.

## 4. Whitepaper contains incorrect equation

Severity: Informational
Type: Undefined Behavior
Target: Whitepaper

Difficulty: High
Finding ID: TOB-UNI-004

**Description**
The whitepaper contains the following statement:

> For example, at any given time, 25% of the assets in a liquidity pool will only be touched if the relative price moves by a factor of 16. (In general, 1 /√ N of the pool's liquidity is only touched if the price moves by a factor of N in one direction.)

*Figure 4.1: Whitepaper, page 1.*

This formula does not make sense, even for a trivial case. When the price is constant (i.e., *N = 1*), the function indicates that 1/1 (i.e., 100%) of the pool's liquidity is touched.

The correct formula is `1- 1 /√ N`.

**Exploit Scenario**
Alice is a Uniswap user or a developer of integrated products. She reads the whitepaper and misunderstands the system, causing her users to lose money.

**Recommendation**
Short term, correct the following sentence:

> For example, at any given time, 75% of the assets in a liquidity pool will only be touched if the relative price moves by a factor of 16. (In general, 1 - 1 /√ N of the pool's liquidity is only touched if the price moves by a factor of N in one direction.)

*Figure 4.2: Corrected version.*

Long term, finalize the whitepaper, ensuring that it is clear.

## 5. Incorrect comparison enables swapping and token draining at no cost

Severity: High
Type: Data Validation
Target: UniswapV3Pool.sol

Difficulty: Low
Finding ID: TOB-UNI-005

**Description**
An incorrect comparison in the `swap` function allows the swap to succeed even if no tokens are paid. This issue could be used to drain any pool of all of its tokens at no cost.

```
// transfer the output
if (amountOut != 0) TransferHelper.safeTransfer(tokenOut, recipient, uint256(-amountOut));

// callback for the input
uint256 balanceBefore = balanceOfToken(tokenIn);
zeroForOne
    ? IUniswapV3SwapCallback(msg.sender).uniswapV3SwapCallback(amountIn, amountOut, data)
    : IUniswapV3SwapCallback(msg.sender).uniswapV3SwapCallback(amountOut, amountIn, data);
require(balanceBefore.add(uint256(amountIn)) >= balanceOfToken(tokenIn), 'IIA');
```

*Figure 5.1: `UniswapV3Pool.sol#L649-L657`*

The `swap` function calculates how many tokens the initiator (`msg.sender`) needs to pay (`amountIn`) to receive the requested amount of tokens (`amountOut`). It then calls the `uniswapV3SwapCallback` function on the initiator's account, passing in the amount of tokens to be paid. The callback function should then transfer at least the requested amount of tokens to the pool contract. Afterward, a `require` inside the `swap` function verifies that the correct amount of tokens (`amountIn`) has been transferred to the pool.

However, the check inside the `require` is incorrect. Instead of checking that *at least* the requested amount of tokens has been transferred to the pool, it checks that *no more than* the requested amount has been transferred. In other words, if the callback does not transfer any tokens to the pool, the check, and the swap, will succeed without the initiator having paid any tokens.

**Exploit Scenario**
Bob deploys a pool for USDT/DAI. The pool holds 1,000,000 DAI. Eve calls a swap for 1,000,000 DAI but transfers 0 USDT, stealing all of the DAI from the pool.

**Recommendation**
Short term, replace the >= with <= inside the `require` in the `swap` function. Add at least one unit test checking that the `IIA` error is thrown when too few tokens are transferred from the initiator's contract to the pool.

Long term, consider adding at least one unit test for each error that can be thrown by the contracts. With a unit test, an error would be thrown when it should be, at least in a simple

case. Also consider adding more properties and using Echidna or Manticore to verify that initiators are correctly transferring tokens to the pool.

## 6. Unbound loop enables denial of service

Severity: Medium
Type: Data Validation
Target: *UniswapV3Pool.sol*

Difficulty: High
Finding ID: TOB-UNI-006

**Description**
The swap function relies on an unbounded loop. An attacker could disrupt swap operations by forcing the loop to go through too many operations, potentially trapping the swap due to a lack of gas.

`UniswapV3Pool.swap` iterates over the tick:

```
    while (state.amountSpecifiedRemaining != 0 && state.sqrtPriceX96 != sqrtPriceLimitX96) {
        StepComputations memory step;

        step.sqrtPriceStartX96 = state.sqrtPriceX96;
        [..]
            state.tick = zeroForOne ? step.tickNext - 1 : step.tickNext;
        } else if (state.sqrtPriceX96 != step.sqrtPriceStartX96) {
            // recompute unless we're on a lower tick boundary (i.e. already transitioned
ticks), and haven't moved
            state.tick = TickMath.getTickAtSqrtRatio(state.sqrtPriceX96);
        }
    }
```

*Figure 6.1: UniswapV3Pool.sol#L544-L619*

On every loop iteration, there is a swap on the current tick's price, increasing it to the next price limit. The next price limit depends on the next tick:

```
    (step.tickNext, step.initialized) = tickBitmap.nextInitializedTickWithinOneWord(
        state.tick,
        tickSpacing,
        zeroForOne
    );

    // ensure that we do not overshoot the min/max tick, as the tick bitmap is not aware of
these bounds
    if (step.tickNext < TickMath.MIN_TICK) {
        step.tickNext = TickMath.MIN_TICK;
    } else if (step.tickNext > TickMath.MAX_TICK) {
        step.tickNext = TickMath.MAX_TICK;
    }

    // get the price for the next tick
```

```
    step.sqrtPriceNextX96 = TickMath.getSqrtRatioAtTick(step.tickNext);
```

*Figure 6.2: `UniswapV3Pool.sol#L549-L563`*

The next tick is the next initialized tick (or an uninitialized tick if no initialized tick is found).

A conservative gas cost analysis of the loop iteration returns the following estimates:

1. ~50,000 gas per iteration if there is no previous fee on the tick (7 SLOAD, 1 SSTORE from non-zero to non-zero, 2 SSTORE from zero to non-zero).
2. ~20,000 gas per iteration if there are previous fees on the tick (7 SLOAD, 3 SSTORE from non-zero to non-zero).

The current block gas limit is 12,500,000. As a result, the swap operation will not be doable if it requires more than 2,500 (scenario 1) or 6,250 (scenario 2) iterations.

An attacker could create thousands of positions with 1 wei to make the system very costly and potentially prevent swap operations.

An attacker would have to pay gas to create the position. However, an Ethereum miner could create a position for free, and if the system were deployed on a layer 2 solution (e.g., optimism), the attacker's gas payment would be significantly lower.

**Exploit Scenario**
Eve is a malicious miner involved with a Uniswap competitor. Eve creates thousands of positions in every Uniswap V3 pool to prevent users from using the system.

**Recommendation**
Short term, to mitigate the issue, determine a reasonable minimum tick spacing requirement, or consider setting a minimum for liquidity per position.

Long term, make sure that all parameters that the owner can enable (such as fee level and tick spacing) have bounds that lead to expected behavior, and clearly document those bounds, such as in a markdown file or in the whitepaper.

## 7. Front-running pool's initialization can lead to draining of liquidity provider's initial deposits

Severity: Medium
Type: Data Validation
Target: *UniswapV3Pool.sol*

Difficulty: High
Finding ID: TOB-UNI-007

**Description**
A front-run on `UniswapV3Pool.initialize` allows an attacker to set an unfair price and to drain assets from the first deposits.

`UniswapV3Pool.initialize` initiates the pool's price:

```
    function initialize(uint160 sqrtPriceX96) external override {
        require(slot0.sqrtPriceX96 == 0, 'AI');

        int24 tick = TickMath.getTickAtSqrtRatio(sqrtPriceX96);

        (uint16 cardinality, uint16 cardinalityNext) =
observations.initialize(_blockTimestamp());

        slot0 = Slot0({
            sqrtPriceX96: sqrtPriceX96,
            tick: tick,
            observationIndex: 0,
            observationCardinality: cardinality,
            observationCardinalityNext: cardinalityNext,
            feeProtocol: 0,
            unlocked: true
        });

        emit Initialize(sqrtPriceX96, tick);
    }
```

*Figure 7.1: UniswapV3Pool.sol#L194-L212*

There are no access controls on the function, so anyone could call it on a deployed pool.

Initializing a pool with an incorrect price allows an attacker to generate profits from the initial liquidity provider's deposits.

**Exploit Scenario**

- Bob deploys a pool for assets A and B through a deployment script. The current market price is 1 A == 1 B.
- Eve front-runs Bob's transaction to the `initialize` function and sets a price such that 1 A ~= 10 B.

- Bob calls mint and deposits assets A and B worth $100,000, sending ~10 times more of asset B than asset A.
- Eve swaps A tokens for B tokens at an unfair price, profiting off of Bob's deployment.

Two tests that demonstrate such an attack are included in Appendix G.

**Recommendation**
Short term, consider
- moving the price operations from `initialize` to the constructor,
- adding access controls to `initialize`, or
- ensuring that the documentation clearly warns users about incorrect initialization.

Long term, avoid initialization outside of the constructor. If that is not possible, ensure that the underlying risks of initialization are documented and properly tested.

## 8. Swapping on zero liquidity allows for control of the pool's price

Severity: Medium | Difficulty: Medium
Type: Data Validation | Finding ID: TOB-UNI-008
Target: `UniswapV3Pool.sol, libraries/SwapMath.sol`

**Description**
Swapping on a tick with zero liquidity enables a user to adjust the price of 1 wei of tokens in any direction. As a result, an attacker could set an arbitrary price at the pool's initialization or if the liquidity providers withdraw all of the liquidity for a short time.

Swapping 1 wei in `exactIn` with a liquidity of zero and a fee enabled will cause `amountRemainingLessFee` and `amountIn` to be zero:

```
uint256 amountRemainingLessFee = FullMath.mulDiv(uint256(amountRemaining), 1e6 - feePips,
1e6);
amountIn = zeroForOne
  ? SqrtPriceMath.getAmount0Delta(sqrtRatioTargetX96, sqrtRatioCurrentX96, liquidity, true)
  : SqrtPriceMath.getAmount1Delta(sqrtRatioCurrentX96, sqrtRatioTargetX96, liquidity, true);
```

*Figure 8.1: `libraries/SwapMath.sol#L41-L44`*

As `amountRemainingLessFee == amountIn`, the next square root ratio will be the square root target ratio:

```
if (amountRemainingLessFee >= amountIn) sqrtRatioNextX96 = sqrtRatioTargetX96;
```

*Figure 8.2: `libraries/SwapMath.sol#L45`*

The next square root ratio assignment results in updates to the pool's price and tick:

```
// shift tick if we reached the next price
if (state.sqrtPriceX96 == step.sqrtPriceNextX96) {
    // if the tick is initialized, run the tick transition
    if (step.initialized) {
        int128 liquidityNet =
            ticks.cross(
                step.tickNext,
                (zeroForOne ? state.feeGrowthGlobalX128 : feeGrowthGlobal0X128),
                (zeroForOne ? feeGrowthGlobal1X128 : state.feeGrowthGlobalX128)
            );
        // if we're moving leftward, we interpret liquidityNet as the opposite sign
        // safe because liquidityNet cannot be type(int128).min
        if (zeroForOne) liquidityNet = -liquidityNet;

        secondsOutside.cross(step.tickNext, tickSpacing, cache.blockTimestamp);
```

```
        state.liquidity = LiquidityMath.addDelta(state.liquidity, liquidityNet);
    }
```

*Figure 8.3: `UniswapV3Pool.sol#L595-L612`*

On a tick without liquidity, anyone could move the price and the tick in any direction. A user could abuse this option to move the initial pool's price (e.g., between its initialization and minting) or to move the pool's price if all the liquidity is temporarily withdrawn.

**Exploit Scenario**

- Bob initializes the pool's price to have a ratio such that 1 token0 == 10 token1.
- Eve changes the pool's price such that 1 token0 == 1 token1.
- Bob adds liquidity to the pool.
- Eve executes a swap and profits off of the unfair price.

Appendix I contains a unit test for this issue.

**Recommendation**

Short term, there does not appear to be a straightforward way to prevent the issue. We recommend investigating the limits associated with pools without liquidity in some ticks and ensuring that users are aware of the risks.

Long term, ensure that pools can never end up in an unexpected state.

## 9. Failed transfer may be overlooked due to lack of contract existence check

Severity: High
Type: Data Validation
Target: `libraries/TransferHelper.sol`

Difficulty: High
Finding ID: TOB-UNI-009

**Description**
Because the pool fails to check that a contract exists, the pool may assume that failed transactions involving destructed tokens are successful.

`TransferHelper.safeTransfer` performs a transfer with a low-level call without confirming the contract's existence:

```
) internal {
    (bool success, bytes memory data) =
        token.call(abi.encodeWithSelector(IERC20Minimal.transfer.selector, to, value));
    require(success && (data.length == 0 || abi.decode(data, (bool))), 'TF');
```

*Figure 9.1: `libraries/TransferHelper.sol#L18-L21`*

The Solidity documentation includes the following warning:

> *The low-level call, delegatecall, and callcode will return success if the calling account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.*

*Figure 9.2: The Solidity documentation details the necessity of executing existence checks prior to performing a delegatecall.*

As a result, if the tokens have not yet been deployed or have been destroyed, `safeTransfer` will return success even though no transfer was executed.

If the token has not yet been deployed, no liquidity can be added. However, if the token has been destroyed, the pool will act as if the assets were sent even though they were not.

**Exploit Scenario**
The pool contains tokens A and B. Token A has a bug, and the contract is destroyed. Bob is not aware of the issue and swaps 1,000 B tokens for A tokens. Bob successfully transfers 1,000 B tokens to the pool but does not receive any A tokens in return. As a result, Bob loses 1,000 B tokens.

**Recommendation**
Short term, check the contract's existence prior to the low-level call in `TransferHelper.safeTransfer`. This will ensure that a swap reverts if the token to be

bought no longer exists, preventing the pool from accepting the token to be sold without returning any tokens in exchange.

Long term, avoid low-level calls. If such a call is not avoidable, carefully review the Solidity documentation, particularly the “Warnings” section.

## 10. `getNextSqrtPriceFromInput|Output` can return a value outside of `MIN_SQRT_RATIO, MAX_SQRT_RATIO`

Severity: Informational | Difficulty: High
Type: Data Validation | Finding ID: TOB-UNI-010
Target: `libraries/SqrtPriceMath.sol, libraries/TickMath.sol`

**Description**

`getNextSqrtPriceFromInput|Output` takes a square price and returns the next square ratio price. A square ratio price should be between [`MIN_SQRT_RATIO`, `MAX_SQRT_RATIO`]; however, `getNextSqrtPriceFromInput|Output` does not confirm that is the case.

The square ratio price's limit is defined with `MIN_SQRT_RATIO/MAX_SQRT_RATIO`:

```
/// @dev The minimum value that can be returned from #getSqrtRatioAtTick. Equivalent to
getSqrtRatioAtTick(MIN_TICK)
uint160 internal constant MIN_SQRT_RATIO = 4295128739;
/// @dev The maximum value that can be returned from #getSqrtRatioAtTick. Equivalent to
getSqrtRatioAtTick(MAX_TICK)
uint160 internal constant MAX_SQRT_RATIO =
1461446703485210103287273052203988822378723970342;
```

*Figure 10.1: `libraries/TickMath.sol#L13-L16`*

`getNextSqrtPriceFromInput/getNextSqrtPriceFromOutput` returns a next square price ratio based on the current one:

```
    /// @notice Gets the next sqrt price given an input amount of token0 or token1
    /// @dev Throws if price or liquidity are 0, or if the next price is out of bounds
    /// @param sqrtPX96 The starting price, i.e., before accounting for the input amount
    /// @param liquidity The amount of usable liquidity
    /// @param amountIn How much of token0, or token1, is being swapped in
    /// @param zeroForOne Whether the amount in is token0 or token1
    /// @return sqrtQX96 The price after adding the input amount to token0 or token1
    function getNextSqrtPriceFromInput(
        uint160 sqrtPX96,
        uint128 liquidity,
        uint256 amountIn,
        bool zeroForOne
    ) internal pure returns (uint160 sqrtQX96) {
        require(sqrtPX96 > 0);
        require(liquidity > 0);

        // round to make sure that we don't pass the target price
        return
            zeroForOne
```

```
                ? getNextSqrtPriceFromAmount0RoundingUp(sqrtPX96, liquidity, amountIn, true)
                : getNextSqrtPriceFromAmount1RoundingDown(sqrtPX96, liquidity, amountIn,
true);
    }

    /// @notice Gets the next sqrt price given an output amount of token0 or token1
    /// @dev Throws if price or liquidity are 0 or the next price is out of bounds
    /// @param sqrtPX96 The starting price before accounting for the output amount
    /// @param liquidity The amount of usable liquidity
    /// @param amountOut How much of token0, or token1, is being swapped out
    /// @param zeroForOne Whether the amount out is token0 or token1
    /// @return sqrtQX96 The price after removing the output amount of token0 or token1
    function getNextSqrtPriceFromOutput(
        uint160 sqrtPX96,
        uint128 liquidity,
        uint256 amountOut,
        bool zeroForOne
    ) internal pure returns (uint160 sqrtQX96) {
        require(sqrtPX96 > 0);
        require(liquidity > 0);

        // round to make sure that we pass the target price
        return
            zeroForOne
                ? getNextSqrtPriceFromAmount1RoundingDown(sqrtPX96, liquidity, amountOut,
false)
                : getNextSqrtPriceFromAmount0RoundingUp(sqrtPX96, liquidity, amountOut,
false);
    }
```

*Figure 10.1: `Libraries/SqrtPriceMath.sol#L102-L146`*

Both functions allow the next square ratio to be outside of its expected bounds.

Currently, the issue is not exploitable, as the bound is checked in `getTickAtSqrtRatio`:

```
   function getTickAtSqrtRatio(uint160 sqrtPriceX96) internal pure returns (int24 tick) {
        // second inequality must be < because the price can never reach the price at the
max tick
        require(sqrtPriceX96 >= MIN_SQRT_RATIO && sqrtPriceX96 < MAX_SQRT_RATIO, 'R');
```

*Figure 10.2: `Libraries/TickMath.sol#L60-L62`*

**Exploit Scenario**

The code is refactored, and the check in `getTickAtSqrtRatio` is removed. `getNextSqrtPriceFromInput` is called with the following and returns 1:

- sqrtPriceX96 = 1925278663495424971823782000289235232968305666l9

- liquidity = 312185657725631617856306979295200193 8
- Amount = 879762240641206834663721924777620520805518046373937138659796718173118496055 29
- Round up = true.

As a result, the next square ratio price is outside of the expected bounds.

**Recommendation**
Short term, check in `getNextSqrtPriceFromInput`/`getNextSqrtPriceFromOutput` that the returned value is within `MIN_SQRT_RATIO`, `MAX_SQRT_RATIO`.

Long term, document every bound for all arithmetic functions and test every bound with Echidna and Manticore.

# A. Vulnerability Classifications

| Vulnerability Classes | |
|---|---|
| **Class** | **Description** |
| Access Controls | Related to authorization of users and assessment of rights. |
| Auditing and Logging | Related to auditing of actions or logging of problems. |
| Authentication | Related to the identification of users. |
| Configuration | Related to security configurations of servers, devices, or software. |
| Cryptography | Related to protecting the privacy or integrity of data. |
| Data Exposure | Related to unintended exposure of sensitive information. |
| Data Validation | Related to improper reliance on the structure or values of data. |
| Denial of Service | Related to causing a system failure. |
| Error Reporting | Related to the reporting of error conditions in a secure fashion. |
| Patching | Related to keeping software up to date. |
| Session Management | Related to the identification of authenticated users. |
| Timing | Related to race conditions, locking, or the order of operations. |
| Undefined Behavior | Related to undefined behavior triggered by the program. |

| Severity Categories | |
|---|---|
| **Severity** | **Description** |
| Informational | The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth. |
| Undetermined | The extent of the risk was not determined during this engagement. |
| Low | The risk is relatively small or is not a risk the customer has indicated is important. |
| Medium | Individual users' information is at risk; exploitation could pose |

| | |
|---|---|
| | reputational, legal, or moderate financial risks to the client. |
| High | The issue could affect numerous users and have serious reputational, legal, or financial implications for the client. |

| **Difficulty Levels** | |
|---|---|
| **Difficulty** | **Description** |
| Undetermined | The difficulty of exploitation was not determined during this engagement. |
| Low | Commonly exploited public tools exist, or such tools can be scripted. |
| Medium | An attacker must write an exploit or will need in-depth knowledge of a complex system. |
| High | An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue. |

## B. Code Maturity Classifications

| Code Maturity Classes | |
|---|---|
| **Category Name** | **Description** |
| Access Controls | Related to the authentication and authorization of components. |
| Arithmetic | Related to the proper use of mathematical operations and semantics. |
| Assembly Use | Related to the use of inline assembly. |
| Centralization | Related to the existence of a single point of failure. |
| Upgradeability | Related to contract upgradeability. |
| Function Composition | Related to separation of the logic into functions with clear purposes. |
| Front-Running | Related to resilience against front-running. |
| Key Management | Related to the existence of proper procedures for key generation, distribution, and access. |
| Monitoring | Related to the use of events and monitoring procedures. |
| Specification | Related to the expected codebase documentation. |
| Testing & Verification | Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.). |

| Rating Criteria | |
|---|---|
| **Rating** | **Description** |
| Strong | The component was reviewed, and no concerns were found. |
| Satisfactory | The component had only minor issues. |
| Moderate | The component had some issues. |
| Weak | The component led to multiple issues; more issues might be present. |
| Missing | The component was missing. |

| | |
|---|---|
| Not Applicable | The component is not applicable. |
| Not Considered | The component was not reviewed. |
| Further Investigation Required | The component requires further investigation. |

# C. Non-security-related Findings

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

**`UniswapV3Pair.sol`:**

- **Prefix emission of an event with the `emit` keyword.** On line 621, the `Swap` event is emitted without using the `emit` keyword. This could confuse readers of the source code. Such an emission would generally cause a solc warning, but no solc warning appears to be raised using solc 0.7.6.

**`SqrtPriceMath.sol`:**

- **Pass in arguments in the correct order.** The first two arguments of the `getAmount0Delta` and `getAmount1Delta` functions are the current and target prices. However, in several places in `SwapMath` and `UniswapV3Pair`, the arguments are passed in reverse (i.e., the input arguments are Q and P, instead of P and Q). This might confuse readers and is not recommended. If this order of arguments is necessary, consider adding a comment above those lines explaining the reasoning.

```
function getAmount0Delta(
    uint160 sqrtPX96, // square root of current price
    uint160 sqrtQX96, // square root of target price
```

*Figure C.1:* `getAmount0Delta` *in* `SqrtPriceMath.sol`.

```
amountOut = SqrtPriceMath.getAmount1Delta(sqrtQX96, sqrtPX96, liquidity, false);
```

*Figure C.2:* `computeSwapStep` *in* `SwapMath.sol#L46`.

```
amountOut = SqrtPriceMath.getAmount0Delta(sqrtQX96, sqrtPX96, liquidity, false);
```

*Figure C.3:* `computeSwapStep` *in* `SwapMath.sol#L52`.

```
amount0 = SqrtPriceMath.getamount0Delta(
    SqrtTickMath.getSqrtRatioAtTick(params.tickUpper).
    slot0.sqrtPriceCurrentX96,
    params.liquidityDelta
);
amount1 = SqrtPriceMath.getamount0Delta(
    SqrtTickMath.getSqrtRatioAtTick(params.tickLower).
    slot0.sqrtPriceCurrentX96,
    params.liquidityDelta
);
```

*Figure C.4:* `_setPosition` *in* `UniswapV3Pair.sol#L430-439`.

- **Rename the `getNextPrice` functions.** The functions `getNextPrice(FromAmount0RoundingUp|Amount1RoundingDown|Input|Output)` return not the price but the root of the price. They should be renamed to `getNextSqrtPriceX`; this will facilitate code comprehension.
- **Fix `isMulSafe`.** Currently, `isMulSafe` throws, consuming all gas, when the first parameter is 0, even though premultiplication by 0 is safe for all unsigned integers, y.

**`BitMath.sol`**

- **Rename the BitMath functions.** The `BitMath.(most|least)SignificantBit` functions don't return the most/least significant bit of an integer. Consider renaming them to better reflect their actual behavior and facilitate code comprehension.
- **Update the `mostSignificantBit`'s comment to contain the correct property.** The comment states that the function returns the following:

```
x >= 2**mostSignificantBit(x) and x < 2**(mostSignificantBit(x)+1)
```

In actuality, it returns the following:

```
x >= 2**mostSignificantBit(x) && (mostSignificantBit(x) == 255 || x <
2**(mostSignificantBit(x)+1))
```

**Documentation**

- **Update the whitepaper.** The whitepaper does not reflect the current state of the codebase. Since the codebase makes extensive use of custom math, it is important to have a clear specification with which the implementation can be compared.

# D. Whitepaper Recommendations

This section lists recommendations for improving the whitepaper (specifically the version provided to Trail of Bits during our initial one-week review in January).

- **Add definitions for each symbol.** Most of the symbols are explained inline. However, some lack a definition. For example, what does $b_0$ stand for— "balance"? Also consider adding a symbol table at the beginning of each section instead of inlining the definitions.
- **Use the same naming system in the whitepaper and the codebase.** The whitepaper includes several references to functions that do not exist in the codebase (e.g., `getValueForPrice`).
- **Add concrete examples of formulas.** Most formulas lack concrete examples, which would help clarify the formulas. For example, consider adding a concrete example of the geometric mean calculation, possibly by comparing it to a concrete example of the arithmetic mean calculation used in previous versions of Uniswap.
- **Replace the TODOs.** There are numerous TODOs throughout the whitepaper. These include small assumptions that still need to be addressed (e.g., in *section 1.5, "do we also assume transfer and transferFrom cause an increase or decrease by the right amount")*, formulas that need to be replaced (e.g., the last line of section 1.5, "Invariants"), and entire sections that need to be added in (e.g., section 1.3, "Crossing a tick"). The inclusion of such a high number of TODOs makes it difficult to fully grasp the system.
- **Add explanations of all concepts to increase the whitepaper's readability.** For example, what exactly does "virtual liquidity" mean?
- **Add more diagrams.** As noted in some of the TODOs, numerous diagrams need to be added. They will help readers visualize the algorithms used.
- **Clearly state which sections are correct and which do not reflect the current state of the code.** While the whitepaper is undergoing revisions, clearly identifying outdated/work-in-progress sections would be highly beneficial to readers.
- **Add cross-tick and within-tick subsections for each level of state.** Section 1.2, "Global state," includes a subsection, 1.2.1, "Swapping within a tick." Also, there is a TODO for subsection 1.3.1, "Crossing a tick," in section 1.3, "Per-tick state." In addition to these two levels of state, the whitepaper includes section 1.4, "Per-position state." Consider adding a subsection for both cross-tick and within-tick swaps in each of these state-level sections, which would help readers fully understand the ticks and how they affect each of the state levels.
- **Add examples of crossing a tick.** One of the trickiest parts of the system is what happens when a tick is crossed. Consider adding extensive examples of crossing a tick.
- **Make the per-level sections subsections of a "state" section.** There are three state levels, each with its own section (1.2, 1.3, and 1.4). Placing all of these under a single section, 1.2, "State," would improve the whitepaper's structure.
- **Use a numbered list for setPosition steps.** Subsection 1.4.1, "setPosition," describes the execution steps for adding or removing liquidity in paragraph form.

Consider using a numbered list to call the readers' attention to these sequential steps.

# E. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

## General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

## ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **`The name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such

cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` and `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

## Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non–token-related functions.** Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

## Owner privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

## Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

# F. Detecting correct `lock` usage

The following contains a Slither script developed during the assessment. We recommend that Uniswap review and integrate the script into its CI.

## Detecting correct `lock` usage

`UniswapV3Pool.lock` has two purposes:

- Preventing the functions from being called before `initialize`
- Preventing reentrancy.

The functions in UniswapV3Pool must have this modifier. The following script follows a whitelist approach, where every reachable function must:

- be protected with the `lock` modifier,
- be whitelisted (including `initialize` and `swap`), or
- be a view function.

No issue was found with the script.

```
from slither import Slither
from slither.core.declarations import Contract
from typing import List


# Init slither
contracts = Slither(
    ".", ignore_compile=True
)  # Remove ignore_compile if the system was not already compiled


def _check_lock(
    contract: Contract, modifiers_access_controls: List[str], whitelist: List[str]
):
    print(f"### Check {contract} lock access controls")
    no_bug_found = True
    for function in contract.functions_entry_points:
        if function.is_constructor:
            continue

        if function.view:
            continue

        if not function.modifiers or (
            not any((str(x) in modifiers_access_controls) for x in function.modifiers)
        ):
```

```
            if not function.name in whitelist:
                print(f"\t- {function.canonical_name} should have a lock modifier")
                no_bug_found = False
    if no_bug_found:
        print("\t- No bug found")


_check_lock(
    contracts.get_contract_from_name("UniswapV3Pool"), ["lock"], ["initialize", "swap"]
)
```

*Figure F.1:* `check-lock.py`

## G. Front-running initialize tests

Below are two tests that show the results of setting a correct vs. incorrect initial price in the `initialize` function. These tests demonstrate that an attacker could abuse TOB-UNI-007 to swap tokens at an unfair price.

```
   it.only('test DAI-USDT -- price 1.0001', async () => {
      // default created pool will use fee 3000, tickSpacing 60

      const init_price = '79228162514264337593543950336'; // sqrtRatio at tick = 0, which is
price 1.00
      const mint_amount = 10000;
      const mint_tickLower = '-887220'; // min for tickSpacing = 60
      const mint_tickUpper = '887220'; // max for tickSpacing = 60

      const swap_amount = 1000
      const swap_priceLimit = '4295128740';

      await pool.initialize(init_price);

      {
        const slot0_after_init = await pool.slot0();
        console.log('pool current tick =', slot0_after_init.tick);
        console.log('pool current price =', slot0_after_init.sqrtPriceX96.toString());
      }

      {
        const t0_bal_before = await token0.balanceOf(pool.address);
        const t1_bal_before = await token1.balanceOf(pool.address);

        console.log(`\nminting ${mint_amount}, with ticks min=${mint_tickLower},
max=${mint_tickUpper}`)
        await mint(wallet.address, mint_tickLower, mint_tickUpper,
expandTo18Decimals(mint_amount))

        const t0_bal_after = await token0.balanceOf(pool.address);
        const t1_bal_after = await token1.balanceOf(pool.address);

        console.log('minter added DAI to pool =',
ethers.utils.formatEther(t0_bal_after.sub(t0_bal_before).toString()))
        console.log('minter added USDT to pool =',
ethers.utils.formatEther(t1_bal_after.sub(t1_bal_before).toString()))
      }

      {
        const swapContractFactory = await ethers.getContractFactory('TestUniswapV3SwapPay')
        const swapContract = (await swapContractFactory.deploy()) as TestUniswapV3SwapPay
        // approve the swap contract to transfer tokens belonging to "wallet"
```

```
        // this is used to pay the required tokens in the uniswapV3SwapCallback
        await token0.approve(swapContract.address, constants.MaxUint256)
        await token1.approve(swapContract.address, constants.MaxUint256)

        const t0_bal_before = await token0.balanceOf(wallet.address);
        const t1_bal_before = await token1.balanceOf(wallet.address);

        console.log(`\nswapping ${swap_amount} DAI for USDT, priceLimit
${swap_priceLimit}`);

        swapContract.swap(pool.address, wallet.address, true, swap_priceLimit,
expandTo18Decimals(swap_amount), expandTo18Decimals(swap_amount), 0)

        const t0_bal_after = await token0.balanceOf(wallet.address);
        const t1_bal_after = await token1.balanceOf(wallet.address);

        const token0_swapper_swap_diff = t0_bal_after.sub(t0_bal_before)
        const token1_swapper_swap_diff = t1_bal_after.sub(t1_bal_before)

        console.log('DAI balance of swapper changed by', token0_swapper_swap_diff.gt('0')
          ? `+${ethers.utils.formatEther(token0_swapper_swap_diff)}`
          : ethers.utils.formatEther(token0_swapper_swap_diff))
        console.log('USDT balance of swapper changed by', token1_swapper_swap_diff.gt('0')
          ? `+${ethers.utils.formatEther(token1_swapper_swap_diff)}`
          : ethers.utils.formatEther(token1_swapper_swap_diff))
      }

      {
        const slot0_after_init = await pool.slot0();
        console.log('\npool current tick =', slot0_after_init.tick);
        console.log('pool current price =', slot0_after_init.sqrtPriceX96.toString());
      }
    })

    it.only('test DAI-USDT -- price 10,000', async () => {
      // default created pool will use fee 3000, tickSpacing 60

      const init_price = '130621891405341611593710811006'; // sqrtRatio at tick = 10000,
which is price 2.71
      const mint_amount = 10000;
      const mint_tickLower = '-887220'; // min for tickSpacing = 60
      const mint_tickUpper = '887220'; // max for tickSpacing = 60

      const swap_amount = 1000
      const swap_priceLimit = '4295128740';

      await pool.initialize(init_price);

      {
```

```
        const slot0_after_init = await pool.slot0();
        console.log('pool current tick =', slot0_after_init.tick);
        console.log('pool current price =', slot0_after_init.sqrtPriceX96.toString());
      }

      {
        const t0_bal_before = await token0.balanceOf(pool.address);
        const t1_bal_before = await token1.balanceOf(pool.address);

        console.log(`\nminting ${mint_amount}, with ticks min=${mint_tickLower},
max=${mint_tickUpper}`)
        await mint(wallet.address, mint_tickLower, mint_tickUpper,
expandTo18Decimals(mint_amount))

        const t0_bal_after = await token0.balanceOf(pool.address);
        const t1_bal_after = await token1.balanceOf(pool.address);

        console.log('minter added DAI to pool =',
ethers.utils.formatEther(t0_bal_after.sub(t0_bal_before).toString()))
        console.log('minter added USDT to pool =',
ethers.utils.formatEther(t1_bal_after.sub(t1_bal_before).toString()))
      }

      {
        const swapContractFactory = await ethers.getContractFactory('TestUniswapV3SwapPay')
        const swapContract = (await swapContractFactory.deploy()) as TestUniswapV3SwapPay
        // approve the swap contract to transfer tokens belonging to "wallet"
        // this is used to pay the required tokens in the uniswapV3SwapCallback
        await token0.approve(swapContract.address, constants.MaxUint256)
        await token1.approve(swapContract.address, constants.MaxUint256)

        const t0_bal_before = await token0.balanceOf(wallet.address);
        const t1_bal_before = await token1.balanceOf(wallet.address);

        console.log(`\nswapping ${swap_amount} DAI for USDT, priceLimit
${swap_priceLimit}`);

        swapContract.swap(pool.address, wallet.address, true, swap_priceLimit,
expandTo18Decimals(swap_amount), expandTo18Decimals(swap_amount), 0)

        const t0_bal_after = await token0.balanceOf(wallet.address);
        const t1_bal_after = await token1.balanceOf(wallet.address);

        const token0_swapper_swap_diff = t0_bal_after.sub(t0_bal_before)
        const token1_swapper_swap_diff = t1_bal_after.sub(t1_bal_before)

        console.log('DAI balance of swapper changed by', token0_swapper_swap_diff.gt('0')
          ? `+${ethers.utils.formatEther(token0_swapper_swap_diff)}`
          : ethers.utils.formatEther(token0_swapper_swap_diff))
```

```
      console.log('USDT balance of swapper changed by', token1_swapper_swap_diff.gt('0')
        ? `+${ethers.utils.formatEther(token1_swapper_swap_diff)}`
        : ethers.utils.formatEther(token1_swapper_swap_diff))
    }

    {
      const slot0_after_init = await pool.slot0();
      console.log('\npool current tick =', slot0_after_init.tick);
      console.log('pool current price =', slot0_after_init.sqrtPriceX96.toString());
    }
  })
```

*Figure H.1: Initialize front-run tests.*

# H. Manual analysis of overflow of amountIn + feeAmount

The following describes our manual analysis of a potential overflow in `UniswapV3Pool.swap`. The overflow is currently not reachable in the system's parameter limits. However, we recommend that Uniswap ensure that the overflow remains unreachable if the parameters are changed.

`SwapMath.computeSwapStep` returns the step's `amountIn` and `feeAmount` (both a uint256):

```
(state.sqrtPriceX96, step.amountIn, step.amountOut, step.feeAmount) =
SwapMath.computeSwapStep(
     state.sqrtPriceX96,
     (zeroForOne ? step.sqrtPriceNextX96 < sqrtPriceLimitX96 : step.sqrtPriceNextX96 >
sqrtPriceLimitX96)
          ? sqrtPriceLimitX96
          : step.sqrtPriceNextX96,
     state.liquidity,
     state.amountSpecifiedRemaining,
     fee
);
```

*Figure I.1: `UniswapV3Pool.sol#L566-L574`*

The variables are added together, `amountIn + feeAmount`, without arithmetic overflow protection:

```
state.amountSpecifiedRemaining -= (step.amountIn + step.feeAmount).toInt256();
[..]
state.amountCalculated = state.amountCalculated.add((step.amountIn +
step.feeAmount).toInt256());
```

*Figure I.2: `UniswapV3Pool.sol#L577-L581`*

We'll show that neither of the above calculations can overflow.

Both variables are computed in `SwapMath.computeSwapStep`. Let's start with `feeAmount`, computed here:

```
        if (exactIn && sqrtRatioNextX96 != sqrtRatioTargetX96) {
            // we didn't reach the target, so take the remainder of the maximum input as fee
            feeAmount = uint256(amountRemaining) - amountIn;
        } else {
            feeAmount = FullMath.mulDivRoundingUp(amountIn, feePips, 1e6 - feePips);
        }
```

*Figure I.3: `SwapMath.sol#L91-L96`*

Because the first case cannot overflow, `feeAmount = amountRemaining - amountIn` (and no underflow happens), `feeAmount + amountIn = amountRemaining`, which by definition fits into a uint256.

Let's consider the second case of the "if" statement. The maximum value of `feePips` is 999,999:

```
require(fee < 1000000);
```

*Figure I.4: UniswapV3Factory.sol#L63*

As such, the maximum value of `feeAmount` is `amountIn * 999,999`.

Now `amountIn` is one of the following (with the X96 suffix removed for readability):

1. `SqrtPriceMath.getAmount0Delta(sqrtRatioA, sqrtRatioB, liquidity, true)`, or
2. `SqrtPriceMath.getAmount1Delta(sqrtRatioA, sqrtRatioB, liquidity, true)`.

Note that `liquidity` is a uint128; let's assume its maximum value is 2^128 - 1. Let's consider these cases separately, as Case 1: getAmount0Delta and Case 2: getAmount1Delta.

## Case 1: getAmount0Delta

```
        return
            roundUp
                ? UnsafeMath.divRoundingUp(
                    FullMath.mulDivRoundingUp(numerator1, numerator2, sqrtRatioBX96),
                    sqrtRatioAX96
                )
```

*Figure I.5: SqrtPriceMath.sol#L164-L169*

`getAmount0Delta` returns (rounding up in the division):

$$\frac{(liquidity * 2^{96}) * (sqrtRatioB - sqrtRatioA)}{sqrtRatioB * sqrtRatioA}$$

Call this expression `E`. We assume that

- `sqrtRatioA` and `sqrtRatioB >= 1` (otherwise, the operation would revert).

So

- `sqrtRatioB <= sqrtRatioB + 1 <= sqrtRatioA * (sqrtRatioB + 1)`
- `sqrtRatioB - sqrtRatioA <= sqrtRatioA * sqrtRatioB`,
- `(sqrtRatioB - sqrtRatioA) / (sqrtRatioB * sqrtRatioA) <= 1`,

which gives us

$$E \leq liquidity * 2^{96} \leq (2^{128} - 1) * 2^{96} < 2^{128} * 2^{96} = 2^{224},$$

resulting in

$$amountIn + feeAmount \leq E + 999{,}999E = 1{,}000{,}000E < 1{,}000{,}000 * 2^{224} < 2^{20} * 2^{224} = 2^{244}.$$

And so

$$amountIn + feeAmount < 2^{244}.$$

It follows that no overflow can happen in this case. Let's move on to the second and final case.

## Case 2: getAmount1Delta

```
        return
            roundUp
                ? FullMath.mulDivRoundingUp(liquidity, sqrtRatioBX96 - sqrtRatioAX96,
FixedPoint96.Q96)
```

*Figure I.6: SqrtPriceMath.sol#L188-L190*

`getAmount1Delta` returns the following (rounding up in the division):

$$\frac{(liquidity) * (sqrtRatioB - sqrtRatioA)}{2^{96}}$$

Call this expression `F`. The ratios are both uint160s; hence their maximum difference is 2^160 - 1. (Note that due to the first line of the function, negative numbers are not possible.)

It follows that

$$F \leq liquidity * (2^{160} - 1) / 2^{96} < liquidity * 2^{(160-96)} = liquidity * 2^{64} < 2^{128} * 2^{64} = 2^{192},$$

resulting in

$$F < 2^{192}.$$

Since this value is even smaller than that in the first case, we can be certain that overflow is not possible in this case either.

# I. Unit test for TOB-UNI-008

The following contains a unit test for TOB-UNI-008, meant to be run in `UniswapV3Pool.spec.ts`.

```
   it.only('test swap 1 wei on empty pool', async () => {
      // default created pool will use fee 3000, tickSpacing 60

      //
      // initialize pool at price 1.0001
      //
      const init_price = '79228162514264337593543950336'; // sqrtRatio at tick = 0, which is
price 1.0001
      await pool.initialize(init_price);

      {
        const slot0_after_init = await pool.slot0();
        console.log('pool current tick =', slot0_after_init.tick);
        console.log('pool current price =', slot0_after_init.sqrtPriceX96.toString());
      }

      {
        const swap_amount = 1; // wei

        // minPrice < priceLimit < currentPrice
        const swap_priceLimit = '4295128740';

        // set up the swap contract
        const swapContractFactory = await ethers.getContractFactory('TestUniswapV3SwapPay')
        const swapContract = (await swapContractFactory.deploy()) as TestUniswapV3SwapPay
        // approve the swap contract to transfer tokens belonging to "wallet"
        // this is used to pay the required tokens in the uniswapV3SwapCallback
        await token0.approve(swapContract.address, constants.MaxUint256)
        await token1.approve(swapContract.address, constants.MaxUint256)

        const t0_bal_before = await token0.balanceOf(wallet.address);
        const t1_bal_before = await token1.balanceOf(wallet.address);
        swapContract.swap(pool.address, wallet.address, true, swap_priceLimit, swap_amount,
swap_amount, 0)
        const t0_bal_after = await token0.balanceOf(wallet.address);
        const t1_bal_after = await token1.balanceOf(wallet.address);
        const t0_diff = t0_bal_after.sub(t0_bal_before)
        const t1_diff = t1_bal_after.sub(t1_bal_before)

        console.log(`\nswapping ${swap_amount} wei token0 for token1, priceLimit
${swap_priceLimit}`);
```

```
        console.log('token0 balance of swapper changed by', t0_diff.gt('0') ?
`+${ethers.utils.formatEther(t0_diff)}` : ethers.utils.formatEther(t0_diff))
        console.log('token1 balance of swapper changed by', t1_diff.gt('0') ?
`+${ethers.utils.formatEther(t1_diff)}` : ethers.utils.formatEther(t1_diff))
      }

      {
        const slot0_after_init = await pool.slot0();
        console.log('\npool current tick =', slot0_after_init.tick);
        console.log('pool current price =', slot0_after_init.sqrtPriceX96.toString());
      }
    })
```

*Figure J.1: Unit test.*