Ramses V3

Date	August 2024
Auditors	Heiko Fisch, Arturo Roura, Vladislav Yaroshuk

1 Executive Summary

This report presents the results of our engagement with **Ramses** to review some of the **Ramses V3** smart contracts.

The review was conducted from **July 2024** to **August 2024** by **Heiko Fisch** and **Arturo Roura**, and extended into **September 2024** by **Vladislav Yaroshuk** and **Arturo Roura**. The extension was due to the discovery of complex vulnerabilities and concerns regarding potential undiscovered issues. During this extended audit, the team focused on analyzing the root causes of these vulnerabilities and evaluating the various solutions proposed by the Ramses team.

The contracts in scope implement a CLMM (Concentrated Liquidity Market Maker) based on Uniswap v3, with several enhancements, including updated functionality for a newer compiler version and dynamic system and protocol fee mechanisms. The Ramses team introduced a new accounting system that tracks the amount of active liquidity each position provides on a weekly basis, recording all weeks a user has contributed liquidity and their share relative to other providers. This system aims to incentivize liquidity provision in pools through an external mechanism that relies on accurate accounting while interacting with the core mechanics.

The codebase lacks sufficient test coverage. After modifying the initial Uniswap v3 codebase, the Ramses team has not adapted their tests, and adding tests will enhance the protocol's security. There are many edge cases in the system, and combined with findings that show underflows and overflows occur unexpectedly, this can lead to undesirable behavior, including reverts during calculations and blocked function executions.

Therefore, while we have reviewed the fixes for the findings described in this report, we recommend thorough and comprehensive testing of these fixes – and the system in general – before considering a production deployment. Ensuring proper test coverage and fuzzing is critical to gain confidence in the contracts' correctness.

2 Scope

Our review focused on the commit hash 061c142c5f53e4d3d19d9caf8b093a837062cc17. The list of files in scope can be found in the Appendix.

3 System overview

Full documentation and a system overview of the Uniswap V3 system can be found in the Uniswap V3 Book.

- **RamsesV3PoolDeployer:**

The contract's main functionality is to deploy a new `RamsesV3Pool`.

- **RamsesV3Factory:**

The contract's main functionality is to create and manage liquidity pools for token trading. This contract includes the `createPool` function, which creates trading pools and can be called by protocol users. It also manages pool fees and protocol fees and has logic to set the `feeCollector` address.

- **RamsesV3Pool:**

This contract serves as the central component of the core contracts, handling key functionalities such as providing and withdrawing liquidity, executing token swaps, and collecting protocol fees. It also features a time-bound mechanism for period calculations and manages the overall state of the system.

- **Oracle:**

This library provides essential functionality for tracking and calculating liquidity and tick data over time. It maintains a list of observations, including timestamps, tick, and liquidity details, allowing it to calculate cumulative values and record observations for new periods. It also computes accumulator values over specific time intervals and calculates the overall cumulative totals. The `snapshotCumulativesInside` function captures and returns a snapshot of cumulative ticks, seconds per liquidity, and seconds within a given tick range, while `periodCumulativesInside` calculates and returns cumulative seconds per liquidity for a specified period and tick range.

- **Tick:**

This library helps manage the state of each tick. It includes functionality for handling liquidity transitions and adjustments as the price moves to different ticks, updating, crossing, and deleting ticks. This library is also used for retrieving fee growth data.

- **Position:**

This library manages the state of each position based on its ticks, and defines various functionalities, such as updating positions, calculating fees, `secondsPerLiquidityInsideX128`, initializing the `secondsPerLiquidityPeriodStartX128` for new periods, and calculating checkpoints.

4 Centralization Risks

A significant deviation in the core logic compared to Uniswap v3: the system is designed to implement dynamic fees, governed by an algorithm that is beyond the scope of this audit. While dynamic fees are a common feature in emerging concentrated liquidity decentralized exchanges, it is important to note that a single address retains the ability to unilaterally modify the fees of a liquidity pool (both the proportion of fees subtracted from each swap and the part of these fees the protocol takes as profit) in a single transaction, without prior notice to users.

5 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

5.1 Actors

The relevant actors are listed below with their respective abilities:

- The Ramses team: The Ramses team has privileged access to manage `RamsesV3Factory` contract, modify pool fees, protocol fees, fee receiver address, enable tick spacing.
- The liquidity providers: Users who willingly provided tokens to the `RamsesV3Pool` contract, making it possible to exchange these tokens in the pool for other users, while liquidity providers receive incentives for that.
- The users: Users who are using provided liquidity to the pools to exchange tokens.

5.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- The tokens which are interacting with the protocol are ERC-20 complaint, works following EIP-20 and are operated correctly.
- The tokens which are interacting with the protocol are not deflationary, the pool won't work correctly with the deflationary tokens.
- The system is correctly initialized with the correct addresses and `initialize` call is sent in the same transaction with the deploy tranasction. After correct initialization, regular pool behaviour is exptected.
- Managers under `restricted` role are trusted, not compromised, and perfomes their roles correctly.

6 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 Debt Is Added Instead of Subtracted in `positionPeriodSecondsInRange` Critical ✓ Fixed

Resolution
Fixed in commit 1f0b7a520f7a9378affab0578d872779ad2562d2

Description

When liquidity is added to or removed from a position, the `secondsDebtX96` member of the reward information gets updated. The intention here is to have a *corrective* value that allows to calculate the period-seconds the position has been in range from its liquidity and period-SPL *at the end of the period*.

More specifically, when liquidity is *added* to a position, we'll later *overestimate* the period-seconds in range, and we're *increasing* the `secondsDebt96`. If, on the other hand, liquidity is *removed* from a position, we'll later *underestimate* the period seconds in range, and `secondsDebt96` is *decreased*:

contracts/CL/core/libraries/Position.sol:L124-L134

```
int256 secondsDebtDeltaX96 = SafeCast.toInt256(
    FullMath.mulDivRoundingUp(
        liquidityDelta > 0 ? uint256(uint128(liquidityDelta)) : uint256(uint128(-liquidityDelta)),
        uint256(uint160(secondsPerLiquidityPeriodIntX128)),
        FixedPoint32.Q32
    )
);

self.periodRewardInfo[period].secondsDebtX96 = liquidityDelta > 0
    ? self.periodRewardInfo[period].secondsDebtX96 + secondsDebtDeltaX96
    : self.periodRewardInfo[period].secondsDebtX96 - secondsDebtDeltaX96; // can't overflow since each period is way less than
```

Based on this logic – positive liquidity delta increases debt, negative liquidity delta decreases debt – the correct thing to do in `positionPeriodSecondsInRange` is *subtracting* `secondsDebtX96` from the product of the positions's liquidity and its period-SPL, both taken at the end of the period. However, what actually happens is that the value is *added*:

contracts/CL/core/libraries/Position.sol:L246-L255

```
periodSecondsInsideX96 = FullMath.mulDiv(liquidity, secondsPerLiquidityInsideX128, FixedPoint32.Q32);

// Need to check if secondsDebtX96>periodSecondsInsideX96, since rounding can cause underflows
if (secondsDebtX96 < 0 || periodSecondsInsideX96 > uint256(secondsDebtX96)) {
    periodSecondsInsideX96 = secondsDebtX96 < 0
        ? periodSecondsInsideX96 - uint256(-secondsDebtX96)
        : periodSecondsInsideX96 + uint256(secondsDebtX96);
} else {
    periodSecondsInsideX96 = 0;
}
```

Note also that the conditon in the `if` statement is the correct one for subtracting, and, finally, that the name "debt" also suggests that the value should be subtracted instead of added. So this is clearly a careless mistake, and not a subtle error, but the consequences are severe: The result returned by `positionPeriodSecondsInRange` is wrong, unless `secondsDebtX96` happens to be 0. As this value is used for the calculation of rewards, this also means that the rewards are generally wrong. Moreover, it is directly exploitable: By starting with a low-liquidity position and adding a lot of liquidity a bit later, thereby creating a huge debt, the period-seconds in range for the position – and therefore the rewards – can be excessively inflated.

Recommendation

The addition has to be changed to a subtraction. Concretely, the two lines

```
                ? periodSecondsInsideX96 - uint256(-secondsDebtX96)
                : periodSecondsInsideX96 + uint256(secondsDebtX96);
```

have to be replaced with

```
                ? periodSecondsInsideX96 + uint256(-secondsDebtX96)
                : periodSecondsInsideX96 - uint256(secondsDebtX96);
```

Since this mistake does not only surface for exotic edge cases but almost always leads to a wrong reward when positions are modified, the presence of this bug also exhibits deficiencies with the test methodology. We strongly recommend implementing a rigorous and comprehensive test methodology before considering a production deployment.

6.2 If `secondsPerLiquidityInsideX128 - secondsPerLiquidityPeriodStartX128` Returns a Negative Value While Minting, Accounting for the Position Will Be Incorrect Major Partially Addressed

Resolution

Partially fixed in commit d214e5858b1618a1d070bc48f0a52caeece10e1a . When a position is newly minted, `initializeSecondStart` records `start` as the current `SPL inside` at that moment to avoid cases where the subtraction described in this issue results in a number below `0` .

We classify this issue as "partially fixed" since the Ramses team has changed the code using our recommendation, however due to complexity of the system this issue still requires further testing to confirm that this fix holds in all of the possible cases. We recommend to increase test coverage, as well as to use fuzzing.

Issue Overview

If we burn a position and then mint it again, if the result of the `secondsPerLiquidityInsideX128 - secondsPerLiquidityPeriodStartX128` in both the `update` function and `positionPeriodSecondsInRange` is **negative**, it will lead to incorrect position accounting.

Issue Description

The `secondsInRange` variable returned by `positionPeriodSecondsInRange` measures how much active liquidity a position provides over a period by multiplying the position's liquidity with the "seconds per liquidity" (SPL) recorded between the two position ticks. This gives a value representing the share of total liquidity contributed by the position in a week. For example, if a position provided all the liquidity for the entire week, `secondsInRange` would equal the total seconds in a week. The following issue addresses an error in accounting for `secondsInside` , which inaccurately tracks the position's provided liquidity during a given period.

The period SPL accounting system that we are reviewing emulates UniswapV3's SPL accounting system, but requires more complexity to calculate the SPL accrued between the two ticks while a position was open in a specific period.

Values to the left of the **start tick** are initialized in the same way as the base SPL accounting system, where they record all SPL accrued **since the start of the period**, and ticks to the right record a `periodSecondsPerLiquidityOutsideX128` (which we will abbreviate to `outside`) of `0` . This is done to correctly account for the SPL accrued by positions that have been minted and in range before the period started.

contracts/CL/core/libraries/Tick.sol:L176-L186

```
if (tick <= periodStartTick && periodSecondsPerLiquidityOutsideBeforeX128 == 0) {
    periodSecondsPerLiquidityOutsideX128 =
        secondsPerLiquidityCumulativeX128 -
        periodSecondsPerLiquidityOutsideBeforeX128 -
        endSecondsPerLiquidityPeriodX128;
} else {
    periodSecondsPerLiquidityOutsideX128 =
        secondsPerLiquidityCumulativeX128 -
        periodSecondsPerLiquidityOutsideBeforeX128;
}
info.periodSecondsPerLiquidityOutsideX128[period] = periodSecondsPerLiquidityOutsideX128;
```

This system still works for the period SPL accounting system, however, since the initialization of the ticks is based on the **start tick** and not the **current tick**, it is perfectly possible to open a position where the `secondsPerLiquidityInsideX128` (which we will abbreviate as `inside`) is **negative**.

These SPL values aren't "correct" by themselves, they don't reflect the amount of SPL accrued between two ticks since the `outside` values are highly dependent on the pool conditions when the ticks were initialized. The only way of securing that `SPL inside` values always represent exactly the amount of SPL accrued between two ticks would be to have initialized **all** the ticks and never reset their `outside` value, therefore recording the exact amount of SPL accrued on each side since the start of the period.

Since this is unfeasible, the system has two different mechanisms to record the difference between the "incorrect" `inside` at the moment of minting a position, and at the moment of querying how much SPL this position has accrued. This is possible since even if the `inside` value is "incorrect"(resulting from the "incorrect" `outside` values registered in a position's ticks), if the tick's `outside` aren't cleared, the system will correctly add the SPL accrued while at the other side of the tick to the "incorrect" values when crossing.

contracts/CL/core/libraries/Tick.sol:L181-L185

```
} else {
    periodSecondsPerLiquidityOutsideX128 =
        secondsPerLiquidityCumulativeX128 -
        periodSecondsPerLiquidityOutsideBeforeX128;
}
```

This means that if we register the `inside` at the moment of minting a position, and the position tick's outside values aren't cleared, we can correctly calculate the difference between the two "incorrect" inside values, to obtain how much SPL was accrued for a position. It is also important to note that since `inside` is relative to the ticks, if several positions hold liquidity in the same ticks they would need a way to register each `inside` value at the moment of minting and burning to account for the exact amount of SPL the position accrued, nothing before and nothing after.

Two mechanisms help keep the accounting correct in this system by doing what was previously described but with different nuances. The main difference is how we account for the SPL the position did or didn't accrue before minting and burning.

The `initializeSecondStart` mechanism:

The `initializeSecondStart` function is called upon the first transaction that modifies the position in the period, and the core of this mechanism is that it registers the `inside` value at that moment in the `secondsPerLiquidityPeriodStartX128` variable(which we will refer to as `start`). Whenever `positionPeriodSecondsInRange` is called to obtain the position `secondsInside`, it will subtract the `start` from the current `inside`, effectively calculating the SPL accrued from the `start` until the current `inside`.

`adjusted inside = current inside - start = (start + net SPL accrued) - start`

If the position held liquidity prior to this, we account for the SPL it has accrued before calling `initializeSecondStart` by registering the position's `secondsInside` at that moment as seconds of debt we will be adding to future `positionPeriodSecondsInRange` calculations.

The mechanism therefore directly affects the way we calculate `periodSecondsInsideX96` before adding or subtracting any debt. This system can only register **negative debt**, meaning seconds that will be added to the final `secondsInRange` calculation.

contracts/CL/core/libraries/Position.sol:L230-L246

```
uint160 secondsPerLiquidityInsideX128 = Oracle.periodCumulativesInside(
    uint32(params.period),
    params.tickLower,
    params.tickUpper,
    params._blockTimestamp
);

// underflow will be protected by sanity check
secondsPerLiquidityInsideX128 = uint160(
    int160(secondsPerLiquidityInsideX128) - secondsPerLiquidityPeriodStartX128
);

RewardInfo storage rewardInfo = states.positions[_positionHash].periodRewardInfo[params.period];
int256 secondsDebtX96 = rewardInfo.secondsDebtX96;

// addDelta checks for under and overflows
periodSecondsInsideX96 = FullMath.mulDiv(liquidity, secondsPerLiquidityInsideX128, FixedPoint32.Q32);
```

Regular Debt mechanism:

The regular debt mechanism instead won't affect the `periodSecondsInsideX96` before adding or subtracting any debt. This mechanism by itself won't manipulate the SPL to reflect the liquidity the position has or hasn't provided between the start of the period and when the `update` was called. It will leave the `inside` value as it is, but will account for whatever liquidity that was or wasn't provided (depending on if minted or burned) purely as debt to be added or subtracted in the final calculation.

Example: For a position that was newly minted at `T1`, and therefore didn't hold any liquidity between `T0` and `T1`, it will calculate the `secondsInside` that the position would have provided if the position was open until `T1` by multiplying the `liquidityDelta` by the `inside` at `T1`, and record those seconds as positive debt.

If we were to call `positionPeriodSecondsInRange` at `T2`, the new increased `inside` would be multiplied by the position's liquidity, resulting in the `secondsInside` the position would have had if the position was held since `T0` until the current `T2`, and then we would subtract the seconds we recorded as debt(from `T0` to `T1`).

These two mechanisms work together to keep track of the correct amount of SPL each position has accrued. Both the `update` function and the `positionPeriodSecondsInRange` will first subtract the `start` from the current `inside`, and then proceed with their execution, in the case of `positionPeriodSecondsInRange` adjusting the `secondsInside` with the debt registered.

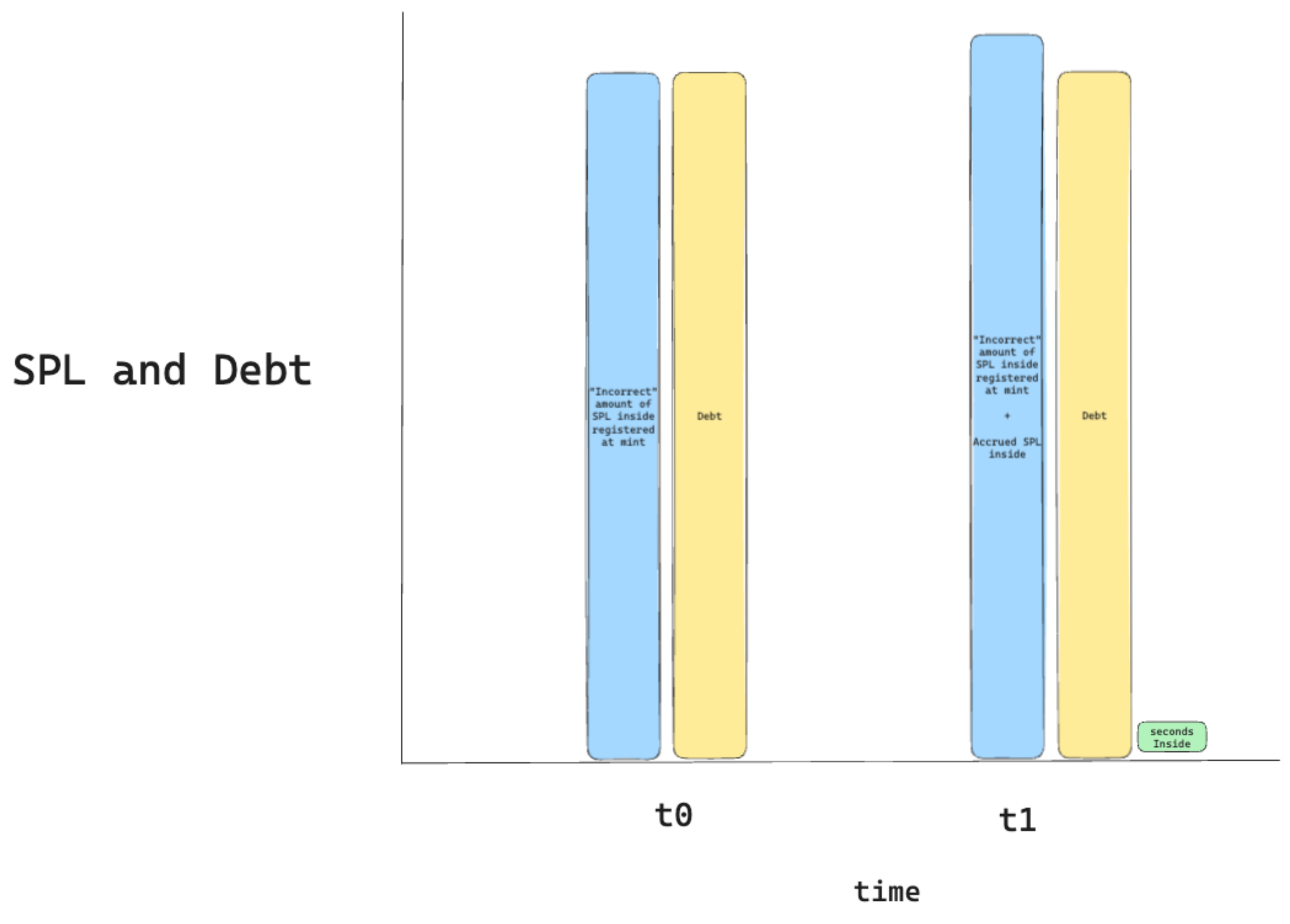
The key moment is the subtraction of the `start` from the `inside`, we will call the resulting number `final inside`:

contracts/CL/core/libraries/Position.sol:L238-L240

```
secondsPerLiquidityInsideX128 = uint160(
    int160(secondsPerLiquidityInsideX128) - secondsPerLiquidityPeriodStartX128
);
```

Previously, we mentioned that minting a position with a negative `inside` is to be expected in this system. For positions with a negative `inside`, the `initializeSecondStart` mechanism will record a negative `start`. All posterior `update` and `positionPeriodSecondsInRange` will subtract the negative `start` from the `inside`, which we know in normal situations can only stay the same or increase. **This means that even if the `inside` is negative, if the `start` is recorded correctly and the `final start` is `0` or positive, the system should calculate the differences in SPL between `start` and new `inside` values correctly.**

Even upon reminting, for positions that are minted with a positive `final inside`, the debt mechanism correctly accounts for whatever amount was not provided between `start` and the new "inaccurate" `SPL inside` value as debt.


SPL and Debt
"Incorrect" amount of SPL inside registered at mint
Debt
"Incorrect" amount of SPL inside registered at mint + Accrued SPL inside
Debt
seconds Inside
t0
t1
time

For this reason, any position that is minted with a negative `SPL inside` relies on this negative `start` value. However, `initializeSecondStart` records the `start` value only once, during the first mint or burn of the period. This is directly tied to the root of this issue: if `secondsPerLiquidityInsideX128 - secondsPerLiquidityPeriodStartX128` results in a negative value, the result will be cast from `int160` to `uint160`, which will result in a number close to `type(uint160).max`. This number will then be multiplied by the liquidity in the position to obtain a huge amount of `periodSecondsInsideX96` that will ultimately be zeroed by the sanity check.

contracts/CL/core/libraries/Position.sol:L257-L260

```
// sanity
if (periodSecondsInsideX96 > 1 weeks * FixedPoint96.Q96) {
    periodSecondsInsideX96 = 0;
}
```

Any position that is burned, and reminted will hold as reference the `start` value from the first minting or burning in the period. The `inside` for the same ticks can also vary greatly if both tick's outside values are cleared or only one of them is cleared and where the current tick is relative to the position ticks while they are minted. If any of these situations make the `final inside` negative while minting, the system will record no debt for the position.

This also means that the accounting will start from the negative `inside` value. If the increasing `inside - start` value ever reaches a positive value, it will start reflecting `periodSecondsInside` from that moment onwards, without counting the SPL between `start` and the `inside` value that made `final inside` positive. If the `final inside` never goes above zero, it will always be an underflowed value of `uint160` and therefore amount to `0` `periodSecondsInside`.

Recommendation

The root of this issue is directly tied to the lack of updating the `secondsPerLiquidityPeriodStartX128` so that it correctly reflects the `secondsPerLiquidityInsideX128`. We, therefore, believe that recording a new `start` value that reflects the current `inside` and then continuing in the same way as usual will avoid making this calculation negative and therefore fixing the issue. Note, though, that we want to keep the debt we have already accumulated, so that shouldn't be reset. We think this works, but it needs heavy testing/fuzzing – which we strongly recommend anyway for a codebase of this complexity.

We also believe that the system could rely completely in the `initializeSecondStart` mechanism to account for all liquidity changes when updating positions, instead of relying on the base debt mechanism. Whenever a position is minted or burned, it would effectively record the `secondsInside` for the last segment(between `start` and the new `inside` value), and would register the resulting seconds as negative debt(to be added to the final result). By then updating the `start`, we prepare the next segment to correctly account for SPL accrued.

The final `positionPeriodSecondsInRange` would consist on the `secondsInside` of the last segment until when a period is finished, adding the seconds of debt representing all previous segments. It's worth mentioning that the `initializeSecondStart` would have to call a modified version of `positionPeriodSecondsInRange` that doesn't add the pre-existing debt, to only account for the `secondsInRange` for the segment. This system would make accounting less problematic, without the need of adding or subtracting possibly huge `secondsDebt` values to adjust the `secondsInside` to the correct value.

A more radical approach would be to prevent the problematic situation in the first place, i.e., if a position has been burned completely, it can't be re-minted in the same period. This restriction is not as severe as it sounds because a user could always choose a different index. Still, it seems less elegant and user-friendly than the solution from the previous paragraph

Examples

Here are two proofs of concept (PoCs):

The first demonstrates how one can obtain a large seconds-in-range value by providing a single second of liquidity in the period, which would ultimately get zeroed out. The second shows how a position that provided liquidity for almost the entire period can receive zero rewards if the tick moves across one of the position ticks while the position is burned and then minted again, which makes the old `start` value be subtracted to a new `inside` of `0`. This underflows the `uint160` creating a large `secondsInside`.

```
beforeEach("initialize the pool at price of 10:1", async () => {
            await pool.initialize(encodePriceSqrt(1n, 10n).toString());
            await mint(wallet.address, 0n, minTick, maxTick, 1n);
        });

        it("SPL underflow", async () => {
            const startTimePeriod1: number = Math.floor(1601906400 / 604800);
            const startTimePeriod2: number = Math.floor((startTimePeriod1 + 1))*604800;
            const deltaTimeToAdvanceForNextPeriod: number = startTimePeriod2-1601906400;
            const newPeriod: number = 1602115200/604800;
            console.log("Current period", newPeriod); //timestamp 1602115200

            await swapToHigherPrice(//to tick 0
                BigInt(79228162514264337593543950336),
                wallet.address,
            );
            await mint(wallet.address, 0n, -60, 0, 200n);


            await pool.advanceTime(deltaTimeToAdvanceForNextPeriod);
            console.log("-------------------------------TEST START-----------------------")
            //NEW PERIOD, TICK is 0
            //tickSpacing is 60

            //In this test we are trying to create an underflow in periodCumulativesInside to register no debt while minting
            //To do so we manipulate the ticks outside crossing the lowerTick twice with one second between crosses to regist
            //We then burn the position and swap so that the currentTick is higher than tickUpper, calculating the periodCumu
            //This test requires the user adds a console log in line 257 of Position.sol to log the seconds inside previous t
            //The test proves that with a single second of liquidity provided, the calculations can result in a large value f


            //Swap to record tickUpper's SPL outside as Cumulative SPL - lastPeriod end SPL, resulting in a really small valu
            await swapToLowerPrice(//to tick -60
            BigInt(78990846045029531151608375686),
            wallet.address,
            );

            //1 second to increase the SPL inside
            await pool.advanceTime(1);

            //Swap to record new Cumulative SPL - (previousCumulativeSPL - lastPeriod end SPL)
            //this results in a higher value than the lastPeriod end SPL
            await swapToHigherPrice(//to tick 0
            BigInt(79228162514264337593543950336),
            wallet.address,
            );
            //We burn the position clearing the tick's `outside` value, they wont have any liquidity which makes ticks not re
            await pool.burn(0, -60, 0, 200n);

            //We swap to a tick under tick lower
            await swapToLowerPrice(//to tick -120
            BigInt(78754240422856966435523493930),
            wallet.address,
            );
            //We advance the time to one second before finishing the period
            await pool.advanceTime(604798);


            //When we mint, its going to calculate the SPL cumulative inside as the lowerTick's SPL outside - the upperTick's
            //This causes an underflow of a uint160 in an unchecked block
            await mint(wallet.address, 0n, -60, 0, 1n);
            //However, the uint160 is used in update, and it is casted to int160, overflowing the cast and resulting in a rea

            //We advance the time to be sure that we are on the next period
            await pool.advanceTime(1);
            //We mint a position in a different ticks just to change the state of the newPeriod
            await mint(wallet.address, 0n, 60, 120, 1n);

            //If you console.log the secondsInside previous to the sanity check, it returns a huge number
            const secondsInRange = await pool.positionPeriodSecondsInRange(2649, wallet.address, 0n, -60, 0);
            console.log("SecondsInRange");
            console.log(secondsInRange);
            console.log("Seconds in a week");
            console.log(604800);

        });
        it.only("No rewards for a position that has been in range", async () => {
            const startTimePeriod1: number = Math.floor(1601906400 / 604800);
            const startTimePeriod2: number = Math.floor((startTimePeriod1 + 1))*604800;
            const deltaTimeToAdvanceForNextPeriod: number = startTimePeriod2-1601906400;
            const newPeriod: number = 1602115200/604800;
            console.log("Current period", newPeriod); //timestamp 1602115200

            await swapToHigherPrice(//to tick 0
                BigInt(79228162514264337593543950336),
                wallet.address,
            );
            await mint(wallet.address, 0n, -60, 0, 200n);


            await pool.advanceTime(deltaTimeToAdvanceForNextPeriod);
            console.log("-------------------------------TEST START-----------------------")
            //NEW PERIOD, TICK is 0
            //tickSpacing is 60

            //We are going to use the same setup as in the SPL underflow test to prove that while a position has held liquidi



            //Swap to record tickUpper's SPL outside as Cumulative SPL - lastPeriod end SPL, resulting in a really small valu
            await swapToLowerPrice(//to tick -60
            BigInt(78990846045029531151608375686),
            wallet.address,
            );

            await pool.advanceTime(604800/2);

            //Swap to record new Cumulative SPL - (previousCumulativeSPL - lastPeriod end SPL)

            await swapToHigherPrice(//to tick 0
            BigInt(79228162514264337593543950336),
            wallet.address,
            );
            //We burn the position clearing the ticks of liquidity, which makes ticks not record crosses
            await pool.burn(0, -60, 0, 200n);

            //We swap to a tick under tick lower
            await swapToLowerPrice(//to tick -120
            BigInt(78754240422856966435523493930),
            wallet.address,
            );



            //When we mint, its going to calculate the SPL cumulative inside as the lowerTick's SPL outside - the upperTick's
            //This causes an underflow of a uint160 in an unchecked block
            await mint(wallet.address, 0n, -60, 0, 200n);
            //We swap back so that the position is in range
            await swapToHigherPrice(//to tick -60
            BigInt(78990846045029531151608375686),
            wallet.address,
            );
            //We advance the time to be sure that we are on the next period
            await pool.advanceTime(604800/2);



            //The position has been in range for most of the period, and returns 0 seconds in range
            const secondsInRange = await pool.positionPeriodSecondsInRange(2649, wallet.address, 0n, -60, 0);
            console.log("SecondsInRange");
            console.log(secondsInRange);
            console.log("Seconds in a week");
            console.log(604800);
        });
```

Remark:

In a discussion with the client during the engagement, we raised concerns whether the system works correctly when ticks are cleared, The code, in its original version, just kept the old value from before the clearing and continued with that when the tick was reactivated. In particular, the `outside` was never reset when a tick was (re-)initialized or cleared. The client recognized this as a mistake and added a commit 274dcd3ca907a51279ab253f70c58f2f80ebc29d that resets `outside` to `0` when a tick is cleared. We were, however, at this point not sure whether this was sufficient or even necessary. In the following investigation, we were able to demonstrate that accounting errors occurred with and without resetting outside, showing that the problem and its solution – which we have explained above – lay deeper than just resetting.

6.3 `Oracle.periodCumulativesInside` Can Return a Wrong Result Major ✓ Fixed

Resolution

It was partially fixed in commit c9ccf442b364a95bdbc1ff73f458a3776292becf. While the `newPeriod` function was modified to make the first second of the period only count towards the new period's state, the period `periodCumulativesInside` was not fixed to adjust the period cap.

Fixed in commit 0e0d40c6f715520ebb179acbebbd229ec9372cda.

Introduction

Uniswap V3 tracks several “outside of a tick amounts,” such as the fee growth or the seconds per liquidity on the other side of the tick. In a very similar way, Ramses V3 also tracks the seconds per liquidity *within a period* that have been accrued on the other side of a tick. Updating this value when a tick is crossed follows the well-known pattern “outside = current - outside”, but the initialization is different in this case. We would like all ticks to the left of the tick at the beginning of a new period to be initialized to the seconds per liquidity (SPL) at the end of the previous period. With this initialization, the familiar formula to compute the “amount between two ticks” can be applied.

However, we can’t really perform this initialization, as it would require iterating over a large number of ticks. Instead, we fix the missing initialization on the fly; whenever we read the “outside amount,” we check whether an adjustment due to the missing initialization is necessary and, if so, use the value we would have received with the proper actual initialization. The adjustment itself is relatively easy (add the SPL at the end of the previous period – the missing initialization), but the question that remains is how to detect whether the adjustment is necessary: We adjust whenever the tick whose “outside amount” we’re reading is (A) to the left of the period’s start tick and (B) the stored “outside amount” at this tick is 0. The idea here is that we’ve never been “that far to the left” (i.e., we never crossed this tick before) in this period and therefore have to fix the missing initialization.

Description

We will argue that there is an edge case in which this logic does not work correctly and leads to a wrong result for the SPL within a period between two ticks. But first, let’s see how the ideas above are applied in actual code. The following is the relevant logic in `Tick.cross`. The “outside amount” we’re interested in is `info.periodSecondsPerLiquidityOutsideX128[period]`, but we’re working with the local variable `periodSecondsPerLiquidityOutsideX128` most of the time. `secondsPerLiquidityCumulativeX128` is the accumulated SPL (“current” in the explanations above), `periodStartTick` is the tick at the start of the period, and `endSecondsPerLiquidityPeriodX128` is the accumulated SPL at the end of the previous period (“initialization value”).

contracts/CL/core/libraries/Tick.sol:L174-L186

```
uint256 periodSecondsPerLiquidityOutsideX128;
uint256 periodSecondsPerLiquidityOutsideBeforeX128 = info.periodSecondsPerLiquidityOutsideX128[period];
if (tick <= periodStartTick && periodSecondsPerLiquidityOutsideBeforeX128 == 0) {
    periodSecondsPerLiquidityOutsideX128 =
        secondsPerLiquidityCumulativeX128 -
        periodSecondsPerLiquidityOutsideBeforeX128 -
        endSecondsPerLiquidityPeriodX128;
} else {
    periodSecondsPerLiquidityOutsideX128 =
        secondsPerLiquidityCumulativeX128 -
        periodSecondsPerLiquidityOutsideBeforeX128;
}
info.periodSecondsPerLiquidityOutsideX128[period] = periodSecondsPerLiquidityOutsideX128;
```

The edge case we want to examine is that `secondsPerLiquidityCumulativeX128` and `endSecondsPerLiquidityPeriodX128` are equal. In this scenario, it is possible to cross a tick and still leave `periodSecondsPerLiquidityOutsideX128` at 0, which will later be interpreted as “we have never been that far left and need to fix the missing initialization.”

Let’s look at a concrete number example: As just discussed, we assume `secondsPerLiquidityCumulativeX128 == endSecondsPerLiquidityPeriodX128`. Let’s say both values are 42. We assume that the current tick, which is also the start tick of the period, is 4, and the swap that is being executed moves the current tick to 2. For all ticks we cross in this process, we set their `periodSecondsPerLiquidityOutsideX128` to 0 (the then-branch in the code above). At the end of this, `periodSecondsPerLiquidityOutsideX128` is (still) 0 for all ticks and, to reiterate, the period’s start tick is 4, and the current tick is 2.

Now assume that we’re still in the same block – and period, hence – and the current tick has not moved in the meantime. Let’s see what happens if `Oracle.periodCumulativesInside` gets called – with the current period, 1 as lower tick, 3 as upper tick, and the current timestamp as arguments. This function is supposed to report the SPL inside the given tick range for the given period; notably, it is called in `Position._updatePosition` (i.e., whenever liquidity is added to or removed from a position), and the result plays an important role for the position accounting.

First, we can see that `Oracle.periodCumulativesInside` checks for both lower and upper tick if a missing-initialization-fix is necessary:

contracts/CL/core/libraries/Oracle.sol:L482-L498

```
int24 startTick = $.periods[period].startTick;
uint256 previousPeriod = $.periods[period].previousPeriod;

snapshot.secondsPerLiquidityOutsideLowerX128 = uint160(lower.periodSecondsPerLiquidityOutsideX128[period]);

if (tickLower <= startTick && snapshot.secondsPerLiquidityOutsideLowerX128 == 0) {
    snapshot.secondsPerLiquidityOutsideLowerX128 = $
        .periods[previousPeriod]
        .endSecondsPerLiquidityPeriodX128;
}

snapshot.secondsPerLiquidityOutsideUpperX128 = uint160(upper.periodSecondsPerLiquidityOutsideX128[period]);
if (tickUpper <= startTick && snapshot.secondsPerLiquidityOutsideUpperX128 == 0) {
    snapshot.secondsPerLiquidityOutsideUpperX128 = $
        .periods[previousPeriod]
        .endSecondsPerLiquidityPeriodX128;
}
```

In our example, since `periodSecondsPerLiquidityOutsideX128` is 0 for tick 1 as well as for tick 3 and both ticks are smaller than the start tick 4, both `snapshot.secondsPerLiquidityOutsideLowerX128` and `snapshot.secondsPerLiquidityOutsideUpperX128` are “fixed” and set to 42. And this is where the mistake happens: We have already crossed tick 3 and applied the missing-initialization-fix then, so we shouldn’t do that again. Going further through the function’s body, we see that we’ll ultimately end up here, where `cache.secondsPerLiquidityCumulativeX128` will be the current SPL, which is still 42:

contracts/CL/core/libraries/Oracle.sol:L539-L542

```
return
    cache.secondsPerLiquidityCumulativeX128 -
    snapshot.secondsPerLiquidityOutsideLowerX128 -
    snapshot.secondsPerLiquidityOutsideUpperX128;
```

So the return value is 42 - 42 - 42 (when it should be 42 - 42 - 0). Taking into account that all three variables have type `uint160` and that we're in an `unchecked` block, we get an underflow and return a very large value instead of 0.

In this example, the `Oracle.periodCumulativesInside` call with the wrong result happens at the very beginning of the period, but it should be noted that the same can happen at any later point in the period: As long as we don't cross tick 3 again, the missing-initialization-fix will be applied a second time and lead to a wrong result.

In summary, we have established that `Oracle.periodCumulativesInside` can return a wrong result, and since this function plays a central role in `Position._updatePosition`, this could mess up our position accounting. However, our initial assumption was that `Tick.cross` gets called when `secondsPerLiquidityCumulativeX128` is equal to `endSecondsPerLiquidityPeriodX128`, and we have to ask ourselves if this can actually happen. The answer is yes, this can happen if the block's timestamp is a multiple of `1 weeks`, which is 604800 (seconds).

The relevant code is in `_advancePeriod` (and, in a very similar form, at the beginning of the `swap` function; we'll just look at `_advancePeriod` here):

contracts/CL/core/RamsesV3Pool.sol:L761-L778

```
// if in new week, record lastTick for previous period
// also record secondsPerLiquidityCumulativeX128 for the start of the new period
uint256 _lastPeriod = $.lastPeriod;
if ((_blockTimestamp() / 1 weeks) != _lastPeriod) {
    Slot0 memory _slot0 = $.slot0;
    uint256 period = _blockTimestamp() / 1 weeks;
    $.lastPeriod = period;

    // start new period in obervations
    uint160 secondsPerLiquidityCumulativeX128 = Oracle.newPeriod(
        $.observations,
        _slot0.observationIndex,
        period
    );

    // record last tick and secondsPerLiquidityCumulativeX128 for old period
    $.periods[_lastPeriod].lastTick = _slot0.tick;
    $.periods[_lastPeriod].endSecondsPerLiquidityPeriodX128 = secondsPerLiquidityCumulativeX128;
```

Note that `period` is defined as `_blockTimestamp() / 1 weeks` and that, in the last line of this excerpt, `endSecondsPerLiquidityPeriodX128` is set to the return value of `Oracle.newPeriod`. This function updates the last observation to time `period * 1 weeks` and returns the SPL at that time:

contracts/CL/core/libraries/Oracle.sol:L336-L349

```
function newPeriod(
    Observation[65535] storage self,
    uint16 index,
    uint256 period
) external returns (uint160 secondsPerLiquidityCumulativeX128) {
    Observation memory last = self[index];
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    unchecked {
        uint32 delta = uint32(period) * 1 weeks - last.blockTimestamp;

        secondsPerLiquidityCumulativeX128 =
            last.secondsPerLiquidityCumulativeX128 +
            ((uint160(delta) << 128) / ($.liquidity > 0 ? $.liquidity : 1));
```

Hence, if `_blockTimestamp() / 1 weeks * 1 weeks` is exactly `_blockTimestamp()` – which is true if and only if `_blockTimestamp()` is a multiple of `1 weeks` – `endSecondsPerLiquidityPeriodX128` is just the same as the SPL at the block's timestamp, which is what we wanted to show.

This concludes the analysis. In the next section, we'll look into possible mitigations.

Recommendation

As discussed above, the mistake happens when, in `Oracle.periodCumulativesInside`, we apply the missing-initialization-fix again for a tick (3) to the left of the start tick (4) that we have already crossed and are currently (2) to the left of. We could try to detect such situations and not apply the fix to a tick T if the current tick (or, more generally, the tick in question – it could also be the tick at the end of a period) is to the left of T. That should work, but a closer look seems to indicate that the root of the problem is something else.

How can the SPL at the end of the previous period be the same as the current SPL – when we're already in the next period? Note that there are no rounding errors involved; the equality stems from using the same time for both SPLs. The problem is that we treat the time `period * 1 weeks` both as the end of the previous period and as the beginning of the current period; in other words, this second is treated as if it belonged to both periods. If we let the previous period end with second `period * 1 weeks - 1`, the problem disappears because the second we use to calculate `endSecondsPerLiquidityPeriodX128` for the previous period is at least one apart from any second in the current period. The only thing we still have to make sure is that no rounding error gets in the way. That is easy to see, though, since the `delta` is first shifted by 128 bits before it is divided by the liquidity, which is a `uint128` and therefore smaller than `1 << 128`.

contracts/CL/core/libraries/Oracle.sol:L349

```
((uint160(delta) << 128) / ($.liquidity > 0 ? $.liquidity : 1));
```

That means, no matter how much liquidity, 1 second will make a difference in the resulting SPL, and the computed `endSecondsPerLiquidityPeriodX128` for the previous period can never be the same as the SPL in the current period. Following this reasoning, our recommendation is to have `Oracle.newPeriod` return the SPL at time `period * 1 weeks - 1`, not at `period * 1 weeks` as is now the case. For clarity, the local variables to which we assign the return value of `Oracle.newPeriod` should probably be renamed to `endSecondsPerLiquidityPeriodX128` or similar:

contracts/CL/core/RamsesV3Pool.sol:L450

```
uint160 secondsPerLiquidityCumulativeX128 = Oracle.newPeriod(
```

contracts/CL/core/RamsesV3Pool.sol:L770

```
uint160 secondsPerLiquidityCumulativeX128 = Oracle.newPeriod(
```

As a side note, the following sanity check should be adjusted too, by appending `- 1` to both occurrences of the expression `currentPeriod * 1 weeks + 1 weeks`:

contracts/CL/core/libraries/Oracle.sol:L520-L523

```
// limit to the end of period
if (cache.time > currentPeriod * 1 weeks + 1 weeks) {
    cache.time = uint32(currentPeriod * 1 weeks + 1 weeks);
}
```

Remark

After-merge timestamps of Ethereum mainnet blocks are odd numbers. Since the difference between two consecutive timestamps is always 12 seconds, that means they will continue to be odd and, therefore, can't be a multiple of the even number `1 weeks`. Hence, the issue described here can't actually occur on Ethereum mainnet.

6.4 If a Period Gets Skipped, `SPL inside` Will Return an Inflated Value for the Period, Which Will Result in Unexpected Behavior Medium Acknowledged

Resolution

The client has acknowledged the issue and recognizes the importance of updating the period state every week for accurate accounting. They have chosen to implement a fix outside the scope of this audit.

Description

When a new period timestamp is reached, the period state is updated to reflect the SPL at the end of the last period(to abbreviate `end SPL`) to perform calculations on the following period and for the past period. The `end SPL` is set to the cumulative SPL until one second before the start of the period.

contracts/CL/core/RamsesV3Pool.sol:L450-L458

```
uint160 secondsPerLiquidityCumulativeX128 = Oracle.newPeriod(
    $.observations,
    slot0Start.observationIndex,
    period
);

// record last tick and secondsPerLiquidityCumulativeX128 for old period
$.periods[_lastPeriod].lastTick = slot0Start.tick;
$.periods[_lastPeriod].endSecondsPerLiquidityPeriodX128 = secondsPerLiquidityCumulativeX128;
```

The period state will only get updated if there is a swap or a position update in the pool. This means that if we start at period `P0` and in the next period `P1` no one where to call any update or swap, and if someone were to swap on `P3` registering a new period, the `end SPL` for `P0` would be the SPL at the end of `P1`.

This is relevant in the `periodCumulativeInside` calculation where if the `lastTick` in the period is between a position's lower and upper tick, it will obtain its SPL inside using as a reference the `end SPL`.

contracts/CL/core/libraries/Oracle.sol:L537-L542

```
    cache.secondsPerLiquidityCumulativeX128 = $.periods[period].endSecondsPerLiquidityPeriodX128;
}
return
    cache.secondsPerLiquidityCumulativeX128 -
    snapshot.secondsPerLiquidityOutsideLowerX128 -
    snapshot.secondsPerLiquidityOutsideUpperX128;
```

This results in inflated values that affect the reward calculation, creating scenarios where users with these kinds of positions get more rewards since their position is accounted as if they provided liquidity from the start of the period until the end of the next period.

This ultimately creates a situation where more rewards can be extracted from the Gauge per period than what was originally allocated for that period. It could also make positions that have held more than half of the active liquidity provided in a period overflow the `secondsInside` sanity check at the end of `periodCumulativeInside`, resulting in no rewards for `P0`.

> Note: The Gauge contract is out of scope.

6.5 Rounding Direction for `secondsDebtDeltaX96` Should Depend on Sign of `liquidityDelta`

Medium ✓ Fixed

Resolution

Fixed in commit 0e0d40c6f715520ebb179acbebbd229ec9372cda.

Description

The debt mechanism to calculate the period-seconds the position has been in range from its liquidity and period-SPL at the end of the period has already been discussed briefly in issue 6.1. As a reminder, when liquidity is *added* to a position, we'll later *overestimate* the period-seconds in range, and we're *increasing* the `secondsDebt96`. If, on the other hand, liquidity is *removed* from a position, we'll later *underestimate* the period seconds in range, and `secondsDebt96` is *decreased*:

contracts/CL/core/libraries/Position.sol:L124-L134

```
int256 secondsDebtDeltaX96 = SafeCast.toInt256(
    FullMath.mulDivRoundingUp(
        liquidityDelta > 0 ? uint256(uint128(liquidityDelta)) : uint256(uint128(-liquidityDelta)),
        uint256(uint160(secondsPerLiquidityPeriodIntX128)),
        FixedPoint32.Q32
    )
);

self.periodRewardInfo[period].secondsDebtX96 = liquidityDelta > 0
    ? self.periodRewardInfo[period].secondsDebtX96 + secondsDebtDeltaX96
    : self.periodRewardInfo[period].secondsDebtX96 - secondsDebtDeltaX96; // can't overflow since each period is way less than
```

As can be seen in the code above, when calculating the `secondsDebtX96Delta`, we're always rounding up. Since rounding should occur in the direction that favors the protocol – which, in our case, means the value `secondsDebtX96` should always be rounded up because it has to be subtracted at the end – a value that is added to `secondsDebtX96` should be rounded up, and a value that is subtracted from `secondsDebtX96` should be rounded down. However, in the code above, the same (rounded-up) value is used for both cases, giving the liquidity provider a rounding advantage.

Recommendation

Rounding in the direction that benefits the user instead of the protocol is dangerous. There have been several cases in the past where a small rounding error could be amplified in some way or other and led to a massive exploit. So even if a rounding error seems small and not amplifiable, we always recommend rounding in the direction that benefits the protocol.

Specifically, in the situation above, `secondsDebtDeltaX96` should be rounded down when `liquidityDelta` is negative.

6.6 Griefing Attack on Deployment of Factory and Deployer Medium ✓ Fixed

Resolution

Fixed in commit 9c46b0d830911c2ce0c5a08c4554b3540cc87681. Access control added to the initialize function.

Description

While Uniswap V3 combines deployer and factory into a single (deployed) contract, due to code size increases and the contract size limit, Ramses V3 deploys deployer and factory separately. More precisely, the steps are as follows:

1. `RamsesV3Factory` is deployed.
2. `RamsesV3PoolDeployer` is deployed. Note that `RamsesV3PoolDeployer`'s constructor takes the address of "its" `RamsesV3Factory` as parameter, so the corresponding factory is known to the deployer.
3. On the deployed `RamsesV3Factory`, `initialize` is called with the address of the `RamsesV3PoolDeployer` as parameter. Now the factory also knows "its" deployer.

Unless these three steps are executed atomically, an attacker could try to call `initialize` on the factory (with a wrong deployer address, obviously) before the Ramses team achieves this. In this case, both the factory and the deployer (if already deployed when the attacker's success is noticed) would be wasted and would have to be redeployed.

Note that is a griefing attack; there is nothing to be gained for an attacker except causing complications and annoyances for the Ramses team.

Recommendation

While this is not a very interesting attack, it can be reliably prevented by executing these three steps atomically, in the same transaction. An alternative option – suggested by a Ramses team member themself – is to make the `initialize` function permissioned; that's a viable solution too.

6.7 Inconsistent `lock` Modifier Minor Partially Addressed

Resolution

Partially fixed in commit 08248a9273fdbc22c1f59178d2124656dc8569ac. Although the `lock` modifier has been added to the `setFee` function, it remains possible to call `advancePeriod` before the pool is initialized. The Ramses team acknowledges this and has chosen to retain the option of not initializing pools upon deployment to accommodate specific use cases.

Description

In the `RamsesV3Pool` contract there is a `lock` modifier, which works not only as reentrancy protection, but also prevents entrance to a function before the pool is initialized. Some of the functions are missing this `lock` modifier without any reason to do that, allowing to call this function while the pool is not initialized. For example the `setFee` function of `RamsesV3Pool` doesn't have `lock` modifier, while a very similar function also related to the fees - `setFeeProtocol` - has this modifier. If the deployer has configured the `sqrtPriceX96` value to be zero, the pool wouldn't be initialized, but a part of its functions will be available and anyone can change storage values of an unitialized pool, for example:

- By calling `_advancePeriod` it's possible to create an initial observation in the pool, while later calling `initialize` function will rewrite this observation back to zero, however it won't rewrite the `_lastPeriod` variable, leading to inconsistency in the system.
- By calling `slot0` view function, the function will return `_slot0.sqrtPriceX96`, `_slot0.tick`, `_slot0.observationCardinality`, `_slot0.observationCardinalityNext` as zeroes, while for example the `MIN_SQRT_RATIO` is `4295128739`, so the zero `sqrtPriceX96` is completely incorrect for the system.
- and other view function will return values for the incorrect pool state.

Consider the following PoC:

```
        it.only("not initialized pool example", async () => {
            // make sure the pool is not initialized
            await expect(
                mint(wallet.address, 0n, -tickSpacing, tickSpacing, 1n),
            ).to.be.revertedWithCustomError(pool, "LOK");

            // calling _advancePeriod for the uninitialized pool works
            await pool._advancePeriod();

            // initialize the pool at price of 10:1, it rewrites the observation
            await pool.initialize(encodePriceSqrt(1n, 10n).toString());

        });
```

Examples

contracts/CL/core/RamsesV3Pool.sol:L758-L787

```
function _advancePeriod() public {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    // if in new week, record lastTick for previous period
    // also record secondsPerLiquidityCumulativeX128 for the start of the new period
    uint256 _lastPeriod = $.lastPeriod;
    if ((_blockTimestamp() / 1 weeks) != _lastPeriod) {
        Slot0 memory _slot0 = $.slot0;
        uint256 period = _blockTimestamp() / 1 weeks;
        $.lastPeriod = period;

        // start new period in obervations
        uint160 secondsPerLiquidityCumulativeX128 = Oracle.newPeriod(
            $.observations,
            _slot0.observationIndex,
            period
        );

        // record last tick and secondsPerLiquidityCumulativeX128 for old period
        $.periods[_lastPeriod].lastTick = _slot0.tick;
        $.periods[_lastPeriod].endSecondsPerLiquidityPeriodX128 = secondsPerLiquidityCumulativeX128;

        // record start tick and secondsPerLiquidityCumulativeX128 for new period
        PeriodInfo memory _newPeriod;

        _newPeriod.previousPeriod = uint32(_lastPeriod);
        _newPeriod.startTick = _slot0.tick;
        $.periods[period] = _newPeriod;
    }
}
```

contracts/CL/core/RamsesV3Pool.sol:L45-L54

```
/// @dev Mutually exclusive reentrancy protection into the pool to/from a method. This method also prevents entrance
/// to a function before the pool is initialized. The reentrancy guard is required throughout the contract because
/// we use balance checks to determine the payment status of interactions such as mint, swap and flash.
modifier lock() {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();
    if (!$.slot0.unlocked) revert LOK();
    $.slot0.unlocked = false;
    _;
    $.slot0.unlocked = true;
}
```

contracts/CL/core/RamsesV3Factory.sol:L56-L87

```
function createPool(
    address tokenA,
    address tokenB,
    int24 tickSpacing,
    uint160 sqrtPriceX96
) external override returns (address pool) {
    require(tokenA != tokenB, IT());
    (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
    require(token0 != address(0), A0());
    uint24 fee = tickSpacingInitialFee[tickSpacing];
    require(fee != 0, F0());
    require(getPool[token0][token1][tickSpacing] == address(0), PE());

    parameters = Parameters({
        factory: address(this),
        token0: token0,
        token1: token1,
        fee: fee,
        tickSpacing: tickSpacing
    });
    pool = IRamsesV3PoolDeployer(RamsesV3PoolDeployer).deploy(token0, token1, tickSpacing);
    delete parameters;

    getPool[token0][token1][tickSpacing] = pool;
    // populate mapping in the reverse direction, deliberate choice to avoid the cost of comparing addresses
    getPool[token1][token0][tickSpacing] = pool;
    emit PoolCreated(token0, token1, fee, tickSpacing, pool);

    if (sqrtPriceX96 > 0) {
        IRamsesV3Pool(pool).initialize(sqrtPriceX96);
    }
}
```

contracts/CL/core/RamsesV3Pool.sol:L739-L741

```
function setFeeProtocol() external override lock {
    ProtocolActions.setFeeProtocol(factory);
}
```

contracts/CL/core/RamsesV3Pool.sol:L754-L756

```
function setFee(uint24 _fee) external override {
    ProtocolActions.setFee(_fee, factory);
}
```

Recommendation

We recommend enforcing initializing pool with the pool deployment.

6.8 Missing Check for Equal Lengths of Arrays Minor ✓ Fixed

Resolution

Fixed in commit 70d153a572e858cadce0cb897537648d9f5a10cf.

Description

contracts/CL/core/RamsesV3Factory.sol:L130-L139

```
function setPoolFeeProtocolBatch(address[] calldata pools, uint8[] calldata _feeProtocols) external restricted {
    for (uint i; i < pools.length; i++) {
        require(_feeProtocols[i] <= 100, FTL());
        uint8 feeProtocolOld = poolFeeProtocol(pools[i]);
        _poolFeeProtocol[pools[i]] = _feeProtocols[i];
        emit SetPoolFeeProtocol(pools[i], feeProtocolOld, feeProtocolOld, _feeProtocols[i], _feeProtocols[i]);

        IRamsesV3Pool(pools[i]).setFeeProtocol();
    }
}
```

The two input arrays for `setPoolFeeProtocolBatch` should have the same length. There is, however, no check for that. If `_feeProtocols` is shorter than `pools`, then a panic will be thrown, which – according to the Solidity documentation – should be avoided. If `_feeProtocols` is longer than `pools`, this is a mistake that probably shouldn't go unnoticed. Hence, we recommend verifying that both arrays have the same length.

contracts/CL/core/RamsesV3Factory.sol:L142-L150

```
function poolFeeProtocol(address pool) public view override returns (uint8 __poolFeeProtocol) {
    __poolFeeProtocol = _poolFeeProtocol[pool];

    if (__poolFeeProtocol == 0) {
        __poolFeeProtocol = feeProtocol;
    }

    return __poolFeeProtocol;
}
```

6.9 Inconsistent Function Return Value and Event Emission Minor ✓ Fixed

Resolution

Fixed in commit 9b718a7dec8bbc0d00f1beef64900739298db980.

In addition to creating new pools with the help of `RamsesV3PoolDeployer`, the main functionality of `RamsesV3Factory` is fee management, especially regarding the protocol fee. There is a default protocol fee for pools, but it is also possible to set a pool-specific protocol fee, which then overrides the default. In the following, we list some minor or informational points regarding the factory and specifically the functionality around the fees.

contracts/CL/core/interfaces/IRamsesV3Factory.sol:L95-L96

```
/// @notice returns the protocol fee for both tokens of a pool.
function poolFeeProtocol(address pool) external view returns (uint8);
```

contracts/CL/core/RamsesV3Factory.sol:L141-L150

```
/// @inheritdoc IRamsesV3Factory
function poolFeeProtocol(address pool) public view override returns (uint8 __poolFeeProtocol) {
    __poolFeeProtocol = _poolFeeProtocol[pool];

    if (__poolFeeProtocol == 0) {
        __poolFeeProtocol = feeProtocol;
    }

    return __poolFeeProtocol;
}
```

The old fee value is, in all three functions, retrieved with the `poolFeeProtocol` function, which returns the default protocol fee if no pool-specific fee has been set. On the other hand, if the new protocol fee is 0 – which means the default protocol fee should be used – the value 0 is emitted for the new fee, not the default fee. Hence, this behavior is inconsistent regarding the old and new fee.

Would it make sense to mimic the behavior for the old value also for the new one, i.e., emit the default protocol fee if the new pool-specific value is 0? – We don't think so. If we did that, note that we couldn't reconstruct the current protocol fee for a pool from the events alone because if the emitted new value is the same as default value, it is unclear if the pool-specific value was coincidentally set to the same value as the default or to 0 and we just *emit* the default value. When the default value changes later, we can't tell whether this pool is affected by this change or not.

Hence, the behavior for the new value shouldn't be changed, but for better consistency, we recommend emitting as old value(s) the actual value set for the pool, i.e., 0 if no pool-specific value has been set (instead of the default). As a general rule, we think it makes the most sense to emit the "raw" value and leave it to the event consumers to infer that a value of 0 means that the default protocol fee applies instead.

6.10 Inaccurate Use of Interface Minor ✓ Fixed

Resolution

Fixed in commit cc9f2c1d2c99f57c775f1d51d61253a0268e414d.

Description

During the deployment of a pool the `parameters` are defined in the factory contract, which subsequently calls the deployer contract to deploy the pool. The pool based on UniV3 design calls the `msg.sender` in the constructor to fetch the `parameters` for itself, which would originally be the factory contract.

contracts/CL/core/RamsesV3Pool.sol:L62-L68

```
constructor() {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    (factory, token0, token1, $.fee, tickSpacing) = IRamsesV3Factory(msg.sender).parameters();

    maxLiquidityPerTick = Tick.tickSpacingToMaxLiquidityPerTick(tickSpacing);
}
```

In the Ramses design, the `parameters` struct is defined in the factory contract, but the deployer has been separated from the factory and now is a different contract. Since the pool will be calling the deployer, to provide it with the `parameters` Ramses creates a new function defined in the deployer contract which fetches the `parameters` from the factory, which makes the deployer act as a middleman for these values.

As you can see in the constructor, in the current version the factory interface is being used to call `msg.sender`, which works because the factory has a function with the same signature defined in it's interface. Although functionally the same, it would be more accurate to use the deployer interface for the call in the pool constructor instead of the factory interface, since it's actually calling the deployer. To do so one should also eliminate the comment block that defines the `parameters` function in the deployer interface, since at the moment it isn't defined.

contracts/CL/core/interfaces/IRamsesV3PoolDeployer.sol:L9-L27

```
/*
/// @notice Get the parameters to be used in constructing the pool, set transiently during pool creation.
/// @dev Called by the pool constructor to fetch the parameters of the pool
/// Returns factory The factory address
/// Returns token0 The first token of the pool by address sort order
/// Returns token1 The second token of the pool by address sort order
/// Returns fee The fee collected upon every swap in the pool, denominated in hundredths of a bip
/// Returns tickSpacing The minimum number of ticks between initialized ticks
function parameters()
    external
    view
    returns (
        address factory,
        address token0,
        address token1,
        uint24 fee,
        int24 tickSpacing
    );
    */
```

Additionally, the NatSpec inheritance for both `parameters()` and `poolBytecode()` is missing, and both are also not present in the interface.

contracts/CL/core/RamsesV3PoolDeployer.sol:L26-L29

```
function parameters()
    external
    view
    returns (address factory, address token0, address token1, uint24 fee, int24 tickSpacing)
```

contracts/CL/core/RamsesV3PoolDeployer.sol:L33-L34

```

function poolBytecode() external pure returns (bytes memory _bytecode) {
```

6.11 No Need to Use Namespaced Storage in Pools Minor ✓ Fixed

Resolution

The client has made a deliberate decision to keep namespaced storage and has added the corresponding NatSpec comment in commit 7c0a92562460fee67a9e5ff595d5d406e57412ad.

Description

The pool contract uses namespaced storage (cf. ERC-7201) instead of the “normal” Solidity storage layout. The former is slightly harder to read, requires some boilerplate code, and most likely increases gas consumption a bit. Namespaced storage is usually associated with the use of proxies, but the Ramses V3 pools are deployed as regular contracts, so namespaced storage has no benefit over the traditional Solidity storage layout.

Recommendation

The namespaced storage is most likely a leftover from a previous version of the codebase that deployed pools as proxies. While we recognize that changing the codebase to use the regular Solidity storage layout will most likely be a tedious task, namespaced storage does have a few minor disadvantages compared to the former, so we recommend considering whether that’s a change worth making.

If you decide to keep namespaced storage, the definition of struct `PoolState`

contracts/CL/core/libraries/PoolStorage.sol:L103-L106

```
// keccak256(abi.encode(uint256(keccak256("pool.storage")) - 1)) & ~bytes32(uint256(0xff));
bytes32 public constant POOL_STORAGE_LOCATION = 0xf047b0c59244a0faf8e48cb6b6fde518e6717176152b6dd953628cd9dccb2800;

struct PoolState {
```

should be preceded with

```
/// @custom:storage-location erc7201:pool.storage
```

according to ERC-7201.

6.12 Optimizations in `RamsesV3Pool`, `Position` ✓ Fixed

Resolution

Fixed in commit 854c086c6ffa798523bc35896aebeac7ba118308.

Description

In the codebase, there are several places where code optimization can improve efficiency and readability:

- In the `_updatePosition` function of the `Position` library, the `_positionHash` variable is calculated using the `positionHash` function. Immediately afterward, the same variables are used again to recalculate `_positionHash` in the `get` function, which is redundant.
- In the `_updatePosition` function of the `Position` library, the `time` variable is declared with the `params._blockTimestamp` value from `calldata`, but during function execution, both `time` and `params._blockTimestamp` are used interchangeably, despite referring to the same value. This creates an additional variable on the stack and increases gas usage unnecessarily.
- In the `_updatePosition` function of the `Position` library, in some places, the `params.tick` variable from `calldata` is used, while in other places, `$.slot0.tick` is used. These variables refer to the same value, creating unnecessary complexity and potential confusion.
- In the `_advancePeriod` function of the `RamsesV3Pool` contract, the `_blockTimestamp() / 1 weeks` calculation is performed twice. In the `swap` function, this calculation is used only once to determine the `period` variable. Declaring the `period` variable earlier in `_advancePeriod` would improve code readability.

Examples

contracts/CL/core/RamsesV3Pool.sol:L431-L445

```
function swap(
    address recipient,
    bool zeroForOne,
    int256 amountSpecified,
    uint160 sqrtPriceLimitX96,
    bytes calldata data
) external override returns (int256 amount0, int256 amount1) {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    uint256 period = _blockTimestamp() / 1 weeks;
    Slot0 memory slot0Start = $.slot0;

    // if in new week, record lastTick for previous period
    // also record secondsPerLiquidityCumulativeX128 for the start of the new period
    uint256 _lastPeriod = $.lastPeriod;
```

contracts/CL/core/RamsesV3Pool.sol:L758-L764

```
function _advancePeriod() public {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    // if in new week, record lastTick for previous period
    // also record secondsPerLiquidityCumulativeX128 for the start of the new period
    uint256 _lastPeriod = $.lastPeriod;
    if ((_blockTimestamp() / 1 weeks) != _lastPeriod) {
```

contracts/CL/core/libraries/Position.sol:L283-L374

```
function _updatePosition(UpdatePositionParams memory params) external returns (PositionInfo storage position) {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    uint256 period = params._blockTimestamp / 1 weeks;

    bytes32 _positionHash = positionHash(params.owner, params.index, params.tickLower, params.tickUpper);

    position = get($.positions, params.owner, params.index, params.tickLower, params.tickUpper);

    uint256 _feeGrowthGlobal0X128 = $.feeGrowthGlobal0X128; // SLOAD for gas optimization
    uint256 _feeGrowthGlobal1X128 = $.feeGrowthGlobal1X128; // SLOAD for gas optimization

    // if we need to update the ticks, do it
    bool flippedLower;
    bool flippedUpper;
    if (params.liquidityDelta != 0) {
        uint32 time = params._blockTimestamp;
        (int56 tickCumulative, uint160 secondsPerLiquidityCumulativeX128) = Oracle.observeSingle(
            $.observations,
            time,
            0,
            $.slot0.tick,
            $.slot0.observationIndex,
            $.liquidity,
            $.slot0.observationCardinality
        );

        flippedLower = Tick.update(
            $._ticks,
            params.tickLower,
            params.tick,
            params.liquidityDelta,
            _feeGrowthGlobal0X128,
            _feeGrowthGlobal1X128,
            secondsPerLiquidityCumulativeX128,
            tickCumulative,
            time,
            false,
            params.maxLiquidityPerTick
        );
        flippedUpper = Tick.update(
            $._ticks,
            params.tickUpper,
            params.tick,
            params.liquidityDelta,
            _feeGrowthGlobal0X128,
            _feeGrowthGlobal1X128,
            secondsPerLiquidityCumulativeX128,
            tickCumulative,
            time,
            true,
            params.maxLiquidityPerTick
        );

        if (flippedLower) {
            TickBitmap.flipTick($.tickBitmap, params.tickLower, params.tickSpacing);
        }
        if (flippedUpper) {
            TickBitmap.flipTick($.tickBitmap, params.tickUpper, params.tickSpacing);
        }
    }

    (uint256 feeGrowthInside0X128, uint256 feeGrowthInside1X128) = Tick.getFeeGrowthInside(
        $._ticks,
        params.tickLower,
        params.tickUpper,
        params.tick,
        _feeGrowthGlobal0X128,
        _feeGrowthGlobal1X128
    );

    uint160 secondsPerLiquidityPeriodX128 = Oracle.periodCumulativesInside(
        uint32(period),
        params.tickLower,
        params.tickUpper,
        params._blockTimestamp
    );

    if (!position.periodRewardInfo[period].initialized) {
        initializeSecondsStart(
            position,
            PositionPeriodSecondsInRangeParams({
                period: period,
                owner: params.owner,
                index: params.index,
                tickLower: params.tickLower,
                tickUpper: params.tickUpper,
                _blockTimestamp: params._blockTimestamp
            }),
            secondsPerLiquidityPeriodX128
        );
    }
```

Recommendation

We recommend refactoring the code in the mentioned locations to save gas, improve code readability, and maintain a clean codebase.

6.13 Typos and Dead Code in the Codebase ✓ Fixed

Resolution

Fixed in commit b2c9a5692d2da2509e7372fd062a5048cc5aa0ff.

Description

In the codebase there are several typos:

- `// start new period in obervations` instead of `// start new period in observations` in `_advancePeriod` function.
- `// start new period in obervations` instead of `// start new period in observations` in `swap` function.

And there is dead code:

- In `balance0` and `balance1` functions of `RamsesV3Pool` there is a commented implementation of low-level call, because prior to `0.8.10` the compiler inserted extra code, including `EXTCODESIZE` (**100 gas**), to check for contract existence for external function calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence. So as for now the low-level implementation is redundant, but the dead code is still present.
- In the `Position` library there is dead code `//PositionInfo memory _self = self;`.

Examples

contracts/CL/core/libraries/Position.sol:L58-L68

```
function update(
    PositionInfo storage self,
    int128 liquidityDelta,
    uint256 feeGrowthInside0X128,
    uint256 feeGrowthInside1X128,
    bytes32 _positionHash,
    uint256 period,
    uint160 secondsPerLiquidityPeriodX128
) internal {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();
    //PositionInfo memory _self = self;
```

contracts/CL/core/RamsesV3Pool.sol:L85-L108

```
function balance0() internal view returns (uint256) {
    /*
    (bool success, bytes memory data) = token0.staticcall(
        abi.encodeWithSelector(IERC20Minimal.balanceOf.selector, address(this))
    );
    require(success && data.length >= 32);
    return abi.decode(data, (uint256));
    */
    return IERC20(token0).balanceOf(address(this));
}

/// @dev Get the pool's balance of token1
/// @dev This function is gas optimized to avoid a redundant extcodesize check in addition to the returndatasize
/// check
function balance1() internal view returns (uint256) {
    /*
    (bool success, bytes memory data) = token1.staticcall(
        abi.encodeWithSelector(IERC20Minimal.balanceOf.selector, address(this))
    );
    require(success && data.length >= 32);
    return abi.decode(data, (uint256));
    */
    return IERC20(token1).balanceOf(address(this));
}
```

contracts/CL/core/RamsesV3Pool.sol:L449

```
// start new period in obervations
```

contracts/CL/core/RamsesV3Pool.sol:L769

```
// start new period in obervations
```

Recommendation

We recommend removing dead code and fixing typos to keep the codebase clean.

6.14 Pools Are Always Deployed With Uninitialized `feeProtocol` Acknowledged

Description

In the `RamsesV3Factory` contract there is `feeProtocol` variable which represents the standard feeProtocol from all of the pools, at the same time there is a `_poolFeeProtocol` `mapping` which stores specific fee protocol if set. However in the `createPool` function of the `RamsesV3Factory` none of these variables are used and the pool is always initialized with the zero `feeProtocol`, and it will use this zero `feeProtocol` untill the manager doesn't call the `setFeeProtocol` and update this variable.

Examples

contracts/CL/core/RamsesV3Factory.sol:L45

```
feeProtocol = 17;
```

contracts/CL/core/RamsesV3Factory.sol:L56-L87

```
function createPool(
    address tokenA,
    address tokenB,
    int24 tickSpacing,
    uint160 sqrtPriceX96
) external override returns (address pool) {
    require(tokenA != tokenB, IT());
    (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
    require(token0 != address(0), A0());
    uint24 fee = tickSpacingInitialFee[tickSpacing];
    require(fee != 0, F0());
    require(getPool[token0][token1][tickSpacing] == address(0), PE());

    parameters = Parameters({
        factory: address(this),
        token0: token0,
        token1: token1,
        fee: fee,
        tickSpacing: tickSpacing
    });
    pool = IRamsesV3PoolDeployer(RamsesV3PoolDeployer).deploy(token0, token1, tickSpacing);
    delete parameters;

    getPool[token0][token1][tickSpacing] = pool;
    // populate mapping in the reverse direction, deliberate choice to avoid the cost of comparing addresses
    getPool[token1][token0][tickSpacing] = pool;
    emit PoolCreated(token0, token1, fee, tickSpacing, pool);

    if (sqrtPriceX96 > 0) {
        IRamsesV3Pool(pool).initialize(sqrtPriceX96);
    }
}
```

contracts/CL/core/RamsesV3Pool.sol:L164-L186

```
function initialize(uint160 sqrtPriceX96) external {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    if ($.slot0.sqrtPriceX96 != 0) revert AI();

    int24 tick = TickMath.getTickAtSqrtRatio(sqrtPriceX96);

    (uint16 cardinality, uint16 cardinalityNext) = Oracle.initialize($.observations, 0);

    _advancePeriod();

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

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

contracts/CL/core/RamsesV3Pool.sol:L738-L741

```
/// @inheritdoc IRamsesV3PoolOwnerActions
function setFeeProtocol() external override lock {
    ProtocolActions.setFeeProtocol(factory);
}
```

Recommendation

We recommend reviewing this scenario, and if it was supposed that every pool is deployed with the `feeProtocol` which equals to the `feeProtocol` of the `RamsesV3Factory`, the `feeProtocol` variable should be added to the initialization.

6.15 `enableTickSpacing` Should Be External ✓ Fixed

Resolution

Fixed in commit b8c02209acba2fa25c5a8e3163874d92ab7ee1e1.

Description

The `enableTickSpacing` of the `RamsesV3Factory` contact is `public` and has the `restricted` modifier, while this function is never used internally and it doesn't lead to any problems except occupying extra bytecode compared to the `external`, this is a bad practice to mark `public` function as `restricted`, if there is no clear reason to do so. Based on the `AccessManaged` contract documentation:

```
 * IMPORTANT: The `restricted` modifier should never be used on `internal` functions, judiciously used in `public`
 * functions, and ideally only used in `external` functions. See {restricted}.
```

Examples

contracts/CL/core/RamsesV3Factory.sol:L90-L100

```
function enableTickSpacing(int24 tickSpacing, uint24 initialFee) public override restricted {
    require(initialFee < 1000000);
    // tick spacing is capped at 16384 to prevent the situation where tickSpacing is so large that
    // TickBitmap#nextInitializedTickWithinOneWord overflows int24 container from a valid tick
    // 16384 ticks represents a >5x price change with ticks of 1 bips
    require(tickSpacing > 0 && tickSpacing < 16384);
    require(tickSpacingInitialFee[tickSpacing] == 0);

    tickSpacingInitialFee[tickSpacing] = initialFee;
    emit TickSpacingEnabled(tickSpacing, initialFee);
}
```

Recommendation

We recommend changing the visibility of the `enableTickSpacing` function from `public` to `external` and use `restricted` with caution.

6.16 `parameters` Struct Should Be Refactored Acknowledged

Description

In the `createPool` function of the `RamsesV3Factory` contract, there is an initialization of the `parameters` struct, which is located in storage and occupies 3 storage slots. During the `IRamsesV3PoolDeployer(RamsesV3PoolDeployer).deploy()` call, the `RamsesV3PoolDeployer` contract passes all of the initialization values to the pool except for the `parameters` struct. The `RamsesV3Pool` contract then calls back the `RamsesV3PoolDeployer.parameters()`, which in turn calls `IRamsesV3Factory(RamsesV3Factory).parameters()` to get the values from the `RamsesV3Factory` storage. Such function execution is redundantly overcomplicated and not gas-optimized, since, right after the `IRamsesV3PoolDeployer(RamsesV3PoolDeployer).deploy()`, the initialized `parameters` struct is deleted, which makes the storage `parameters` variable operate like a memory variable but with extra cost. Additionally, the `parameters` view function of the `RamsesV3PoolDeployer` contract will always return an empty struct for other callers, as the `parameters` are immediately cleared from the `RamsesV3Factory` contract.

Examples

contracts/CL/core/RamsesV3PoolDeployer.sol:L21-L32

```
function deploy(address token0, address token1, int24 tickSpacing) external returns (address pool) {
    require(msg.sender == RamsesV3Factory);
    pool = address(new RamsesV3Pool{salt: keccak256(abi.encodePacked(token0, token1, tickSpacing))}());
}

function parameters()
    external
    view
    returns (address factory, address token0, address token1, uint24 fee, int24 tickSpacing)
{
    (factory, token0, token1, fee, tickSpacing) = IRamsesV3Factory(RamsesV3Factory).parameters();
}
```

contracts/CL/core/RamsesV3Pool.sol:L62-L68

```
constructor() {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();

    (factory, token0, token1, $.fee, tickSpacing) = IRamsesV3Factory(msg.sender).parameters();

    maxLiquidityPerTick = Tick.tickSpacingToMaxLiquidityPerTick(tickSpacing);
}
```

contracts/CL/core/RamsesV3Factory.sol:L27-L35

```
struct Parameters {
    address factory;
    address token0;
    address token1;
    uint24 fee;
    int24 tickSpacing;
}

Parameters public parameters;
```

contracts/CL/core/RamsesV3Factory.sol:L56-L77

```
function createPool(
    address tokenA,
    address tokenB,
    int24 tickSpacing,
    uint160 sqrtPriceX96
) external override returns (address pool) {
    require(tokenA != tokenB, IT());
    (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
    require(token0 != address(0), A0());
    uint24 fee = tickSpacingInitialFee[tickSpacing];
    require(fee != 0, F0());
    require(getPool[token0][token1][tickSpacing] == address(0), PE());

    parameters = Parameters({
        factory: address(this),
        token0: token0,
        token1: token1,
        fee: fee,
        tickSpacing: tickSpacing
    });
    pool = IRamsesV3PoolDeployer(RamsesV3PoolDeployer).deploy(token0, token1, tickSpacing);
    delete parameters;
```

Recommendation

We recommend refactoring the `parameters` struct, passing it with other variables to the `deploy` function and then to the `constructor` to make the code cleaner and more gas-efficient, removing the `RamsesV3PoolDeployer.parameters()` function.

6.17 Inconsistent `setFeeProtocol` Comment ✓ Fixed

Resolution

Fixed in commit 5b1bb1210f08d703924e680397839b056cc8f931.

Description

In the `RamsesV3Factory` contract in the `constructor` the `feeProtocol` is initialized to be `17`, which is equal to 17%, while in the `setFeeProtocol` function of `ProtocolActions` library there is a comment:

```
    /// @notice Set the protocol's % share of the fees
    /// @dev Fees start at 50%, with 5% increments
```

Which is not accurate and differs what the value with which the `RamsesV3Factory` contract is initialized.

Examples

contracts/CL/core/RamsesV3Factory.sol:L37-L45

```
constructor(address accessManager) AccessManaged(accessManager) {
    tickSpacingInitialFee[10] = 500;
    emit TickSpacingEnabled(10, 500);
    tickSpacingInitialFee[60] = 3000;
    emit TickSpacingEnabled(60, 3000);
    tickSpacingInitialFee[200] = 10000;
    emit TickSpacingEnabled(200, 10000);

    feeProtocol = 17;
```

contracts/CL/core/libraries/ProtocolActions.sol:L27-L29

```
/// @notice Set the protocol's % share of the fees
/// @dev Fees start at 50%, with 5% increments
function setFeeProtocol(address factory) external {
```

Recommendation

We recommend clarifying with which value the pool should be initialized, and changing comments of `setFeeProtocol` function.

6.18 Unused Imports in `RamsesV3Pool` ✓ Fixed

Resolution

Fixed in commit 5ee93d80fc7f909f953e971efbdb664bf7c9be64.

Description

In the `RamsesV3Pool` contract imports the `IRamsesV3PoolDeployer` interface which is never used and the contract can be compiled without it. Also `IRamsesV3PoolImmutables` and `IRamsesV3PoolState` interfaces are never used even for NatSpec `@inheritdoc`, which also can be removed

Example

contracts/CL/core/RamsesV3Pool.sol:L4

```
import {IRamsesV3PoolImmutables, IRamsesV3PoolState, IRamsesV3PoolActions, IRamsesV3PoolDerivedState, IRamsesV3PoolOwnerAction
```

contracts/CL/core/RamsesV3Pool.sol:L19

```
import {IRamsesV3PoolDeployer} from './interfaces/IRamsesV3PoolDeployer.sol';
```

Recommendation

We recommend removing unused imports to keep the codebase clean.

6.19 Burn of the Unexisting Positions Reverts With Panic Code Acknowledged

Description

In the `burn` function of `UniswapV3Pool` contract there is no validation that the position which is burned has any liquidity, or has been created before. Because of that such function execution will consume a lot of gas from the caller and revert late with the panic error either in the `update` call in the `Tick` library, when the `liquidityDelta` is subtracted from `liquidity` in case there is no liquidity, or even later in the `update` call of the `Position` library, since the `liquidity` of the position is zero. Consider following PoC:

```

    it.only("burn overflows with panic code 0x11", async () => {
        // before adding liquidity to the pool
        await expect(pool.connect(wallet).burn(0, -120, 0, BigInt(3000))).to.be.revertedWithPanic();

        await mint(other.address, 0n, -120, 0, BigInt(3000));

        // after adding liquidity to the pool
        await expect(pool.connect(wallet).burn(0, -120, 0, BigInt(3000))).to.be.revertedWithPanic();
    });
```

Examples

contracts/CL/core/RamsesV3Pool.sol:L343-L358

```
function burn(
    uint256 index,
    int24 tickLower,
    int24 tickUpper,
    uint128 amount
) external override lock advancePeriod returns (uint256 amount0, uint256 amount1) {
    unchecked {
        (PositionInfo storage position, int256 amount0Int, int256 amount1Int) = _modifyPosition(
            ModifyPositionParams({
                owner: msg.sender,
                index: index,
                tickLower: tickLower,
                tickUpper: tickUpper,
                liquidityDelta: -int256(uint256(amount)).toInt128()
            })
        );
```

contracts/CL/core/libraries/Tick.sol:L92-L110

```
function update(
    mapping(int24 => TickInfo) storage self,
    int24 tick,
    int24 tickCurrent,
    int128 liquidityDelta,
    uint256 feeGrowthGlobal0X128,
    uint256 feeGrowthGlobal1X128,
    uint160 secondsPerLiquidityCumulativeX128,
    int56 tickCumulative,
    uint32 time,
    bool upper,
    uint128 maxLiquidity
) internal returns (bool flipped) {
    TickInfo storage info = self[tick];

    uint128 liquidityGrossBefore = info.liquidityGross;
    uint128 liquidityGrossAfter = liquidityDelta < 0
        ? liquidityGrossBefore - uint128(-liquidityDelta)
        : liquidityGrossBefore + uint128(liquidityDelta);
```

contracts/CL/core/libraries/Position.sol:L58-L80

```
function update(
    PositionInfo storage self,
    int128 liquidityDelta,
    uint256 feeGrowthInside0X128,
    uint256 feeGrowthInside1X128,
    bytes32 _positionHash,
    uint256 period,
    uint160 secondsPerLiquidityPeriodX128
) internal {
    PoolStorage.PoolState storage $ = PoolStorage.getStorage();
    //PositionInfo memory _self = self;

    uint128 liquidity = self.liquidity;
    uint128 liquidityNext;

    if (liquidityDelta == 0) {
        if (liquidity <= 0) revert NP(); // disallow pokes for 0 liquidity positions
        liquidityNext = liquidity;
    } else {
        liquidityNext = liquidityDelta < 0
            ? liquidity - uint128(-liquidityDelta)
            : liquidity + uint128(liquidityDelta);
    }
```

Recommendation

We recommend to add validation to the `burn` function and revert early in case the position doesn't have liquidity or hasn't been created, and also to provide better logging of the revert message.

6.20 Minor Inconsistency in Variable Naming for Storage Root ✓ Fixed

Resolution

Fixed in commit 605fd3261f14b8205d3c1eb3fcaaf0df19b8009b.

Description

Pools use namespaced storage (also see https://github.com/ConsenSysDiligence/ramses-v3-audit-2024-07/issues/5), and when a function needs to access the pool's storage it usually starts with

```
PoolStorage.PoolState storage $ = PoolStorage.getStorage();
```

In particular, the use of the variable `$` for the storage root is canoncial in the codebase. There is one exception, though, where the variable `states` is used instead:

contracts/CL/core/libraries/Position.sol:L200-L203

```
function positionPeriodSecondsInRange(
    PositionPeriodSecondsInRangeParams memory params
) public view returns (uint256 periodSecondsInsideX96) {
    PoolStorage.PoolState storage states = PoolStorage.getStorage();
```

Recommendation

For better readability and consistent naming, we recommend using `$` also in `positionPeriodSecondsInRange`.

6.21 Several `require` Statements Without Error ✓ Fixed

Resolution

Fixed in commit f16bb3f6f7587f227582c04b7a34cebd89d96f1e.

Description

Several `require` statements in the codebase do not have an error or error message. Errors provide helpful information in case of reverts and help with debugging.

(e.g., `RamsesV3Factory.initialize`, `RamsesV3Factory.enableTickSpacing`, `RamsesV3PoolDeployer.deploy`).

Examples

contracts/CL/core/RamsesV3Factory.sol:L51

```
require(RamsesV3PoolDeployer == address(0));
```

contracts/CL/core/RamsesV3Factory.sol:L91

```
require(initialFee < 1000000);
```

contracts/CL/core/RamsesV3Factory.sol:L95-L96

```
require(tickSpacing > 0 && tickSpacing < 16384);
require(tickSpacingInitialFee[tickSpacing] == 0);
```

contracts/CL/core/RamsesV3PoolDeployer.sol:L22

```
require(msg.sender == RamsesV3Factory);
```

Recommendation

Consider adding an error to each `require` statement that doesn't have one.

6.22 Minor Issues in Factory and Factory Interface ✓ Fixed

Resolution

The client has fixed these issues and has continued polishing the NatSpec on other contracts and interfaces.

Description

The NatSpec annotations in the Factory contract are inherited from the interface, but there are instances where there is no NatSpec for a function, where the inheritance is missing, where the Natspec is incorrect and where the function doesn't appear in the interface.

A. There is an inconsistency with the emission of events. Unlike the set*FeeProtocol* functions, setFee does not emit an event on the factory contract It should be noted, though, that there is no corresponding state in the factory, and the call is just forwarded to the pool – where an event is emitted. But then again, for the protocol fee, an event is emitted on both the pool and the factory, so one might still want to add an event for fee changes to the factory.

B.

contracts/CL/core/interfaces/IRamsesV3Factory.sol:L27-L30

```
/// @notice Emitted when a new fee amount is enabled for pool creation via the factory
/// @param tickSpacing The minimum number of ticks between initialized ticks
/// @param fee The fee, denominated in hundredths of a bip
event TickSpacingEnabled(int24 indexed tickSpacing, uint24 indexed fee);
```

The NatSpec emphasizes on the fee being enabled, which was the key value in Uni v3, and doesn't mention tick spacing. Since in Ramses tick spacing is the key and fees are dynamic, the NatSpec should emphasize that the tick spacing is enabled.

C.

contracts/CL/core/interfaces/IRamsesV3Factory.sol:L117-L118

```

function setFee(address _pool, uint24 _fee) external;
```

Missing NatSpec in the Interface and missing inheritance in the Factory.

contracts/CL/core/RamsesV3Factory.sol:L156-L160

```

    function setFee(address _pool, uint24 _fee) external override restricted {
        IRamsesV3Pool(_pool).setFee(_fee);
    }
}
```

D.

contracts/CL/core/RamsesV3Factory.sol:L24-L25

```

address public feeCollector;
```

NatSpec described in the Interface, missing inheritance.

E.

contracts/CL/core/RamsesV3Factory.sol:L34-L35

```

Parameters public parameters;
```

NatSpec described in the Interface, missing inheritance.

F.

contracts/CL/core/RamsesV3Factory.sol:L109-L110

```
function setPoolFeeProtocol(address pool, uint8 _feeProtocol) external restricted {
```

Function not present in the interface and missing Natspec.

G.

contracts/CL/core/RamsesV3Factory.sol:L118-L119

```
function setPoolFeeProtocolBatch(address[] calldata pools, uint8 _feeProtocol) external restricted {
```

Function not present in the interface and missing Natspec.

H.

contracts/CL/core/RamsesV3Factory.sol:L129-L130

```
function setPoolFeeProtocolBatch(address[] calldata pools, uint8[] calldata _feeProtocols) external restricted {
```

Function not present in the interface and missing Natspec.

I.

contracts/CL/core/RamsesV3Factory.sol:L151-L152

```
function setFeeCollector(address _feeCollector) external override restricted {
```

NatSpec described in the interface, missing inheritance.

J.

contracts/CL/core/RamsesV3Factory.sol:L142-L150

```
function poolFeeProtocol(address pool) public view override returns (uint8 __poolFeeProtocol) {
    __poolFeeProtocol = _poolFeeProtocol[pool];

    if (__poolFeeProtocol == 0) {
        __poolFeeProtocol = feeProtocol;
    }

    return __poolFeeProtocol;
}
```

The `return` statement at the end of the function body is not necessary, since `__poolFeeProtocol` is returned anyway.

K.

contracts/CL/core/interfaces/IRamsesV3Factory.sol:L37-L49

```
/// @notice Emitted when the protocol fee is changed
/// @param pool The pool address
/// @param feeProtocol0Old The previous value of the token0 protocol fee
/// @param feeProtocol1Old The previous value of the token1 protocol fee
/// @param feeProtocol0New The updated value of the token0 protocol fee
/// @param feeProtocol1New The updated value of the token1 protocol fee
event SetPoolFeeProtocol(
    address pool,
    uint8 feeProtocol0Old,
    uint8 feeProtocol1Old,
    uint8 feeProtocol0New,
    uint8 feeProtocol1New
);
```

The functions `setPoolFeeProtocol` and both variants of `setPoolFeeProtocolBatch` emit the event `SetPoolFeeProtocol`, defined in `IRamsesV3Factory`. It is unclear why the event has separate values for the two tokens because the fee is always the same for both tokens.

L.

The auto-generated getter for the public state variable `RamsesV3PoolDeployer` is not declared in the interface.

6.23 Unused Import in `PoolStorage.sol` ✓ Fixed

Resolution

Fixed in commit 784cb39594fdb32d9ceb3419ac2a037ec17812ef.

Description

`PoolStorage.sol` imports `IERC20Minimal`:

contracts/CL/core/libraries/PoolStorage.sol:L4

```
import {IERC20Minimal} from './../interfaces/IERC20Minimal.sol';
```

However, this interface was only used in the functions `balance0` and `balance1`, which have been put inside a `/* ... */` comment:

contracts/CL/core/libraries/PoolStorage.sol:L130-L156

```
/*
/// @dev Get the pool's balance of token0
/// @dev This function is gas optimized to avoid a redundant extcodesize check in addition to the returndatasize
/// check
function balance0() internal view returns (uint256) {
    PoolState storage states = getStorage();

    (bool success, bytes memory data) = states.token0.staticcall(
        abi.encodeWithSelector(IERC20Minimal.balanceOf.selector, address(this))
    );
    require(success && data.length >= 32);
    return abi.decode(data, (uint256));
}

/// @dev Get the pool's balance of token1
/// @dev This function is gas optimized to avoid a redundant extcodesize check in addition to the returndatasize
/// check
function balance1() internal view returns (uint256) {
    PoolState storage states = getStorage();

    (bool success, bytes memory data) = states.token1.staticcall(
        abi.encodeWithSelector(IERC20Minimal.balanceOf.selector, address(this))
    );
    require(success && data.length >= 32);
    return abi.decode(data, (uint256));
}
*/
```

Recommendation

As `balance0` and `balance1` are now implemented in `RamsesV3Pool`, their old implementations in comments can be removed from `PoolStorage`, and so can the import that is not needed anymore.

6.24 Superfluous Subtraction of Variable With Value Zero ✓ Fixed

Resolution

Fixed in commit 33acc3492f4bb7ad3a4fcf18bb064e8e723608ed.

Description

When a tick is crossed, `periodSecondsPerLiquidityOutsideX128` for the current period has to be updated (along with other `Outside` variables familiar from Uniswap V3):

contracts/CL/core/libraries/Tick.sol:L174-L186

```
uint256 periodSecondsPerLiquidityOutsideX128;
uint256 periodSecondsPerLiquidityOutsideBeforeX128 = info.periodSecondsPerLiquidityOutsideX128[period];
if (tick <= periodStartTick && periodSecondsPerLiquidityOutsideBeforeX128 == 0) {
    periodSecondsPerLiquidityOutsideX128 =
        secondsPerLiquidityCumulativeX128 -
        periodSecondsPerLiquidityOutsideBeforeX128 -
        endSecondsPerLiquidityPeriodX128;
} else {
    periodSecondsPerLiquidityOutsideX128 =
        secondsPerLiquidityCumulativeX128 -
        periodSecondsPerLiquidityOutsideBeforeX128;
}
info.periodSecondsPerLiquidityOutsideX128[period] = periodSecondsPerLiquidityOutsideX128;
```

It should be noted that in the then-branch `periodSecondsPerLiquidityOutsideBeforeX128` is 0, so it doesn't have to be subtracted.

Recommendation

The following line can be removed:

contracts/CL/core/libraries/Tick.sol:L179

```
periodSecondsPerLiquidityOutsideBeforeX128 -
```

6.25 New Struct Fields `endSecondsPerLiquidityPeriodX128` and `periodStartTick` Would Be a Better Fit for `SwapCache` Than `SwapState` Acknowledged

Resolution

Acknowledged by the client.

Description

During swaps, both Uniswap V3 and Ramses V3 utilize a memory variable `cache` of struct type `SwapCache` and a memory variable `state` of struct type `SwapState`. The `cache` variable is supposed to hold data that doesn't change during execution of the swap (such as the liquidity at the beginning of the swap or the current timestamp), while the data in `state` changes as the swap progresses (such as the amounts already swapped or the current tick).

The definitions of these structs are as follows:

contracts/CL/core/RamsesV3Pool.sol:L374-L411

```
struct SwapCache {
    // the protocol fee for the input token
    uint8 feeProtocol;
    // liquidity at the beginning of the swap
    uint128 liquidityStart;
    // the timestamp of the current block
    uint32 blockTimestamp;
    // the current value of the tick accumulator, computed only if we cross an initialized tick
    int56 tickCumulative;
    // the current value of seconds per liquidity accumulator, computed only if we cross an initialized tick
    uint160 secondsPerLiquidityCumulativeX128;
    // whether we've computed and cached the above two accumulators
    bool computedLatestObservation;
    // timestamp of the previous period
    uint32 previousPeriod;
}

// the top level state of the swap, the results of which are recorded in storage at the end
struct SwapState {
    // the amount remaining to be swapped in/out of the input/output asset
    int256 amountSpecifiedRemaining;
    // the amount already swapped out/in of the output/input asset
    int256 amountCalculated;
    // current sqrt(price)
    uint160 sqrtPriceX96;
    // the tick associated with the current price
    int24 tick;
    // the global fee growth of the input token
    uint256 feeGrowthGlobalX128;
    // amount of input token paid as protocol fee
    uint128 protocolFee;
    // the current liquidity in range
    uint128 liquidity;
    // seconds per liquidity at the end of the previous period
    uint256 endSecondsPerLiquidityPeriodX128;
    // starting tick of the current period
    int24 periodStartTick;
}
```

Compared to Uniswap, Ramses adds

- one new field to `StructCache`: `previousPeriod`, which stores the previous period;
- two new fields to `StructState`: `endSecondsPerLiquidityPeriodX128`, which holds seconds per unit of liquidity at the end of the previous period, and `periodStartTick`, which stores the tick at the beginning of the current period.

However, neither of these change during the execution of a swap, so they would all be a better fit for `StructCache`. In addition to that, `cache.previousPeriod` is only used once, when `state.endSecondsPerLiquidityPeriodX128` is written, so it might be removed.

contracts/CL/core/RamsesV3Pool.sol:L500

```
endSecondsPerLiquidityPeriodX128: $.periods[cache.previousPeriod].endSecondsPerLiquidityPeriodX128,
```

Recommendation

As argued above, it might be conceptually more appropriate to move `endSecondsPerLiquidityPeriodX128` and `periodStartTick` from the `SwapState` struct to `SwapCache` and then remove `previousPeriod` from `SwapCache`. However, we do realize that this would likely lead to a lengthy expression (`$.periods[$.periods[period].previousPeriod].endSecondsPerLiquidityPeriodX128`) for the assignment of `endSecondsPerLiquidityPeriodX128` (or to bigger code changes), and this is most likely the reason for the placement of the new field members in the first place. While we personally prefer the conceptually cleaner approach (and accept the lengthy expression as price to pay), we recognize that this is a judgment call and offer these thoughts merely for your consideration.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts/CL/core/RamsesV3Factory.sol	`01255a3f90cb6fed0e923c44042f7b69a6e9be50`
contracts/CL/core/RamsesV3Pool.sol	`6a68f765238d0166f45da657b10eecee8b7eb43e`
contracts/CL/core/RamsesV3PoolDeployer.sol	`3e8304443a90c3b324bb5eda450919e28fefdb7d`
contracts/CL/core/libraries/Oracle.sol	`9e5fda60ae1d66fca83e76c4b4236c39d18008e6`
contracts/CL/core/libraries/Position.sol	`98b4c906a56e02b3322e9c7f4afb75142999a58b`

File	SHA-1 hash
contracts/CL/core/libraries/Tick.sol	`66508eca5ba7c67e556e6e523cd562d2aaffd7d4`
contracts/CL/core/libraries/PoolStorage.sol	`abc0a8a4760efec34b48da5339e6ae1861dbdae6`

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.