Security Review Report
NM-0180 JediSwap

NETHERMIND
SECURITY

(May 6, 2024)

Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind for the JediSwap AMM implementation. JediSwap is a concentrated liquidity AMM designed for Starknet, written in the Cairo language. The JediSwap implementation is a translation of Uniswap V3 written in Cairo, including the core and periphery contracts. The protocol had already been launched on Starknet mainnet at the beginning of the audit engagement, with some issues affecting certain users and positions. For this reason, the engagement was split into two distinct sections: triaging known mainnet issues and then inspecting the code as part of the standard audit.

The audit was performed using: (a) manual analysis of the codebase, (b) simulation of the smart contracts, (c) analyzing historical interactions with the protocol, and (d) comparative testing against the Solidity-equivalent implementation.

Along this document, we report 14 points of attention, where two are classified as `Critical`, one is classified as `High` and ten are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 describes the process of exploring the issues present on the mainnet. Section 5 describes the differential testing with custom tools. Section 6 discusses the risk rating methodology adopted for this audit. Section 7 details the issues. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, and automated tests. Section 10 concludes the document.

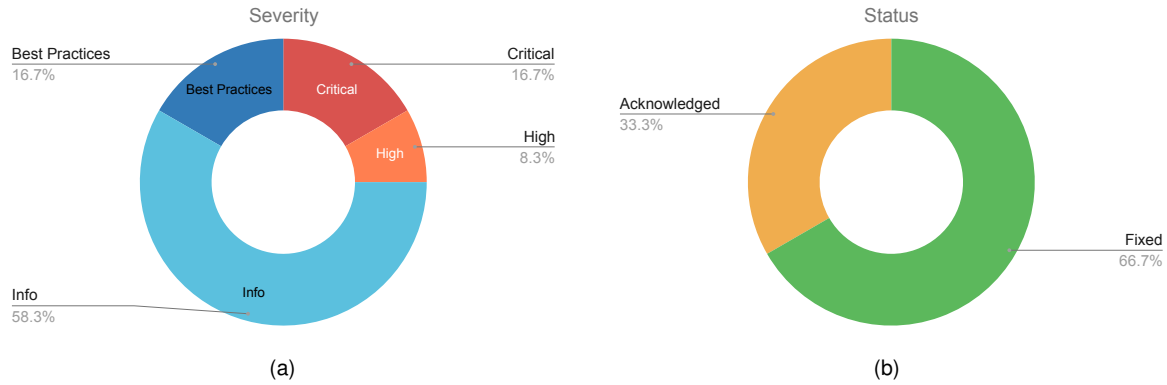

Fig. 1: Distribution of issues: Critical (2), **High** (1), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (7), **Best Practices** (2). **Distribution of status: Fixed** (8), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review, Issue Triage
Initial Report	April 16, 2024
Response from Client	Continuous communication
Final Report	May 6, 2024
Repository	jediswaplabs/JediSwap-v2-core jediswaplabs/JediSwap-v2-periphery
Commit (Audit)	`20e0bb9d9c2ecf30d4d3f32697a609c8c120d950` `fa83957cc5dea02f495047f8aeff238313931ec1`
Commit (Final)	`33526cb72d22d9bd5fc324f410493d41db49d75b` `8a2e445d8b8f5f7351dd19444c1e0957fb2929f4`
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

`JediSwap-v2-core` repository:

	Contract	LoC	Comments	Ratio	Blank	Total
1	jediswap_v2_pool.cairo	840	185	22.0%	121	1146
2	lib.cairo	16	0	0.0%	2	18
3	jediswap_v2_factory.cairo	162	69	42.6%	41	272
4	libraries/position.cairo	106	30	28.3%	17	153
5	libraries/tick_bitmap.cairo	93	26	28.0%	11	130
6	libraries/tick.cairo	176	46	26.1%	24	246
7	libraries/tick_math.cairo	217	19	8.8%	44	280
8	libraries/sqrt_price_math.cairo	164	65	39.6%	18	247
9	libraries/swap_math.cairo	111	15	13.5%	10	136
	Total	**1885**	**455**	**24.1%**	**288**	**2628**

`JediSwap-v2-periphery` repository:

	Contract	LoC	Comments	Ratio	Blank	Total
1	jediswap_v2_swap_router.cairo	346	42	12.1%	45	433
2	jediswap_v2_nft_position_manager.cairo	637	169	26.5%	97	903
3	lib.cairo	11	0	0.0%	2	13
4	libraries/periphery_payments.cairo	17	7	41.2%	1	25
5	libraries/nft_descriptor.cairo	33	4	12.1%	2	39
6	libraries/liquidity_amounts.cairo	94	30	31.9%	10	134
7	libraries/callback_validation.cairo	19	10	52.6%	2	31
	Total	**1157**	**262**	**22.6%**	**159**	**1578**

Contracts added to `JediSwap-v2-core` repository during security review at commit 8de4e924a842fe5111452eb9d512013a91506532:

	Contract	LoC	Comments	Ratio	Blank	Total
1	libraries/bitshift_trait.cairo	61	10	16.4%	10	81
2	libraries/full_math.cairo	35	4	11.4%	4	43
3	libraries/bit_math.cairo	81	12	14.8%	5	98
4	libraries/math_utils.cairo	19	5	26.3%	2	26
5	libraries/signed_integers/i128.cairo	220	82	37.3%	56	358
6	libraries/signed_integers/integer_trait.cairo	4	4	100.0%	0	8
7	libraries/signed_integers/i256.cairo	245	82	33.5%	61	388
8	libraries/signed_integers/i16.cairo	215	82	38.1%	55	352
9	libraries/signed_integers/i32.cairo	240	82	34.2%	60	382
	Total	**1120**	**363**	**32.4%**	**253**	**1736**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect rounding during next tick calculation affects swap	Critical	Fixed
2	Positions with overflowed fees cannot be managed	Critical	Fixed
3	Swaps can fail due to overflow in new price calculations	High	Fixed
4	Bitshift operation `shr` can return off-by-one incorrect value for negative `i256` inputs	Info	Fixed
5	Incorrect output for left bit shift by 128 for negative numbers	Info	Fixed
6	Solidity signed integers don't support `abs`	Info	Fixed
7	Missing overflow functionality on protocol and position fees	Info	Fixed
8	Right bitshifts will revert instead of truncating significant bits	Info	Acknowledged
9	Shifts where `n` is greater than `255` lead to revert	Info	Acknowledged
10	Unused `owner` field in `JediSwapV2Factory`	Info	Fixed
11	Factory upgrade function handles two classhashes	Best Practices	Acknowledged
12	Simplify input for bitshift functions	Best Practices	Acknowledged

4 Mainnet Issues

First mainnet issue

At the start of the audit engagement, the JediSwap team presented a problem that they had been experiencing on Starknet Mainnet. The first week of the engagement was spent triaging this issue. The issue presented itself as positions where the position fee X128 values had underflowed and once underflowed, these users were not able to exit their liquidity positions through the burn function or call collect.

It was first suspected by the JediSwap team that overflowed fee amounts should not be happening, and this was causing the issue. However, the Uniswapv3 implementation does allow for fee amount overflows, as it simply tracks the delta between the global fee growth and the position fee growth. With Solidity 0.7.0 being the language used for Univ3, this works as intended, and the delta can always be found even when overflow has occurred.

However, the Cairo language signed integer types prevent overflows for safety reasons. In some areas of the code, the JediSwap team has used custom functions that will mimic this overflowing behavior, however in one area of the code related to fee delta calculations, they use a standard Cairo unsigned integer subtraction rather than an overflowing subtraction. When a position fee growth had overflowed, the attempted subtraction from the global growth being done with the standing Cairo library integer subtraction caused an overflow and led to transactions being reverted, effectively locking the LP of any user whose position fee growth counters had overflowed.

Second mainnet issue

After the cause of the first issue had been resolved, the JediSwap team revealed a second mainnet issue, where there were liquidity solvency issues. The liquidity pool would run out of tokens before all users are able to exit their positions, leaving some users with LP positions that have no underlying tokens or value. This issue was challenging to diagnose, given Starknet's limited debugging tooling. Some tools were written to analyze the on-chain data to observe where these solvency issues first occurred by simulating every user exiting their position using a binary search approach between the current block and the block where the contract was deployed. Once the first transaction causing this issue had been discovered, we observed that the cause was a rounding issue in the dependency of the yas-core signed integer implementation. It should be noted that in the readme of this document, there is a disclaimer that specifically states the code is not ready for production and has not been audited.

The dependency issue was related to rounding on division for negative numbers. Given that the JediSwap implementation is a re-implementation of the UniswapV3 protocol in the Cairo language, signed integer math should behave the same as the Solidity 0.7 signed integers. The solidity approach is to round nearest to zero every time. If you have a calculation that would end up as -12.7, it is rounded closer to zero, so the result would be -12. The yas-core implementation would round towards zero for positive numbers, but in the negative range, it would round to the nearest integer, so -12.7 would be -13. This difference in tick calculations led to an internal pricing error, which could not be observed in price alone since any slight deviation would be corrected by arbitrage, but the underlying amount of tokens that each position should own is now not correct as it assumes the tick calculations are operating normally.

Due to this second issue, the JediSwap team decided to implement their own signed integer library during the audit engagement.

5 Differential Testing

5.1 Uniswap vs JediSwap

Given that the JediSwap implementation is a translation of Uniswap v3 into the Cairo language, this offered a unique testing opportunity where both implementations can be tested simultaneously, and the results after each interaction can be compared to ensure that the behavior of JediSwap matches Uniswap. The testing framework is designed to have both an Ethereum Anvil node and a Starknet Katana node with Uniswap and JediSwap deployed, respectively. A set of interaction data (mint, burn, collect, liquidity add/remove) is then passed to the framework, where each interaction is converted into compatible calldata for each node, and then the interaction is executed.

After each interaction, invariants are checked such as liquidity position data, tick positioning, amount of tokens transferred in/out, and fees charged. Any discrepancy is flagged, allowing developers to triage the underlying difference in implementation, which caused an invariant to break. To improve the testing data, we have collected and provided real-world data from the following pools on Uniswap v3:

- **WETH-USDC**: `0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640`
- **USDC-USDT**: `0x7858E59e0C01EA06Df3aF3D20aC7B0003275D4Bf`
- **WETH-SHIB**: `0x2f62f2b4c5fcd7570a709dec05d68ea19c82a9ec`

5.2 Solidity vs Cairo Signed Integers

During the security review engagement, the team developed a custom library for signed integers. The library includes signed integer types with basic functions like arithmetic operations and bit shifts. The behavior of the library should be equivalent to the behavior of the numbers and operations in Solidity. Therefore, to test the equivalence of both implementations, the testing framework was developed. The tests provide random data in defined ranges as the input for Cairo and Solidity functions and compare the output. If the output differs or there is an error (e.g., overflow), then it is saved to a JSON file for further analysis.

6 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

7 Issues

7.1 [Critical] Incorrect rounding during next tick calculation affects swap

File(s): yas-core-dependency

Description: The JediSwap project utilized the signed integers from the `yas_core` library. The division for those numbers is implemented in a non-standard way, which differs from how division for signed integers works in Solidity. Below, we present the division function for the `i32`:

```
fn i32_div(a: i32, b: i32) -> i32 {
    i32_check_sign_zero(a);
    // Check that the divisor is not zero.
    assert(b.mag != 0, 'b can not be 0');

    // The sign of the quotient is the XOR of the signs of the operands.
    let sign = a.sign ^ b.sign;

    if (sign == false) {
        // If the operands are positive, the quotient is simply their absolute value quotient.
        return ensure_non_negative_zero(a.mag / b.mag, sign);
    }

    // If the operands have different signs, rounding is necessary.
    // First, check if the quotient is an integer.
    if (a.mag % b.mag == 0) {
        let quotient = a.mag / b.mag;
        if (quotient == 0) {
            return IntegerTrait::new(quotient, false);
        }
        return ensure_non_negative_zero(quotient, sign);
    }

    // If the quotient is not an integer, multiply the dividend by 10 to move the decimal point over.
    let quotient = (a.mag * 10) / b.mag;
    let last_digit = quotient % 10;

    if (quotient == 0) {
        return ensure_non_negative_zero(quotient, false);
    }

    // Check the last digit to determine rounding direction.
    if (last_digit <= 5) {
        return ensure_non_negative_zero(quotient / 10, sign);
    } else {
        return ensure_non_negative_zero((quotient / 10) + 1, sign);
    }
}
```

The division that results in the negative quotient that is not an integer is rounded to the nearest integer. Below, we present examples of such rounding:

```
use yas_core::numbers::signed_integer::i32::i32;
// -2/5 == -0.4 --> 0
IntegerTrait::<i32>::new(2, true) / IntegerTrait::<i32>::new(5, false) == IntegerTrait::<i32>::new(0, false)
// -3/5 == -0.6 --> -1
IntegerTrait::<i32>::new(3, true) / IntegerTrait::<i32>::new(5, false) == IntegerTrait::<i32>::new(1, true)
```

This is not consistent with how the Solidity does the rounding, as presented below:

```
// -2/5 == -0.4 --> 0
 int32(-2)/int32(5) == 0
// -3/5 == -0.6 --> 0
 int32(-3)/int32(5) == 0
```

This rounding difference affects core swap functionality. The `swap(...)` function computes the next tick with `next_initialized_tick_within_one_word(...)`. There, it checks if, within the distance of one word, there is an initialized tick in a bitmap. We present part of this function below:

```
fn next_initialized_tick_within_one_word(
    ref self: ComponentState<TContractState>,
    tick: i32,
    tick_spacing: u32,
    search_left: bool
) -> (i32, bool) {
    let tick_spacing_i = IntegerTrait::<i32>::new(tick_spacing, false);
    let mut compressed: i32 = tick / tick_spacing_i;
    if ((tick < IntegerTrait::<i32>::new(0, false))
        && (tick % tick_spacing_i != IntegerTrait::<i32>::new(0, false))) {
        compressed -= IntegerTrait::<i32>::new(1, false);
    };

    if (search_left) {
        let (word_pos, bit_pos) = position(compressed);
        let word: u256 = self.bitmap.read(word_pos);
        // all the 1s at or to the right of the current bitPos
        let mask: u256 = 1_u256.shl(bit_pos.into()) - 1 + 1_u256.shl(bit_pos.into());
        let masked: u256 = word & mask;

        // if there are no initialized ticks to the right of or at the current tick, return rightmost in the word
        let initialized = masked != 0;
        // overflow/underflow is possible, but prevented externally by limiting both tickSpacing and tick
        let next = if (initialized) {
            (compressed - (bit_pos - BitMath::most_significant_bit(masked)).into())
                * tick_spacing_i
        } else {
            (compressed - bit_pos.into()) * tick_spacing_i
        };
        return (next, initialized);
    }
    // @audit-note: The rest of the code is omitted
    ...
}
```

The rounding issue may cause an incorrect value in the `compressed` variable. In the pool at which we identified the problem, the `compressed` was computed by `-8/10` and resulted in `-1`, which was different from the Solidity implementation where the result would be `0`. The incorrect `compressed` variable is then passed to the `position(...)` function that outputs the incorrect coordinates, which results in the incorrect `word` read from bitmask. In effect, the tick that has liquidity is not found, and the swap process is continued as if that tick does not exist. As a result, the tick and liquidity are not updated. Incorrect liquidity will influence the amount of rewards for the position owner, as well as the final tick after the swap. This issue was discovered by investigating the problem of claiming rewards of position owners. The investigation included fetching mainnet data of the affected pool and replaying the transactions while checking protocol invariants. After narrowing the problem to the `swap(...)` function, we reviewed the code manually, which led us to the problem with rounding on the division of negative signed integers.

Recommendation(s): Consider changing the implementation of division for signed integers so that the rounding is equivalent to Solidity, i.e., always round towards zero.

Status: Fixed

Update from the client: Wrote our own integers based on `yas_core` and changed the incorrect code.

7.2 [Critical] Positions with overflowed fees cannot be managed

File(s): JediSwap-v2-periphery/jediswap_v2_nft_position_manager.cairo

Description: On Starknet mainnet, it was observed that some liquidity pool positions were not able to be managed, where any attempt to collect fees or increase/decrease liquidity on these affected positions would lead to the transaction reverting. After analyzing the failed transaction calldata and collecting on-chain data related to the position, we determined that the revert was caused by an overflow on a `u256`, in the periphery "position manager" contract. We identified that this revert occurred in the following code pattern:

```
mul_div(
    fee_growth_inside_0_X128 - position_info.fee_growth_inside_0_last_X128,
    position_info.liquidity.into(),
    Q128
)
```

This code pattern is present in the functions `collect`, `increase_liquidity`, and `decrease_liquidity`. It calculates the difference in fee growth since it had last been checked and then scales those newly earned rewards by the amount of liquidity present in the position. Under most circumstances, the subtraction will work correctly, however it is possible for a position's fee growth to overflow, an intentional design choice in the source implementation. This design choice works since even if an overflow occurs, the delta is consistent between the global and position fee growth. A snippet from the function `TickComponent::get_fee_growth_inside` shows that in the JediSwap implementation, overflows are supported:

```
(
    mod_subtraction(
        mod_subtraction(fee_growth_global_0_X128, fee_growth_below_0_X128),
        fee_growth_above_0_X128
    ),
    mod_subtraction(
        mod_subtraction(fee_growth_global_1_X128, fee_growth_below_1_X128),
        fee_growth_above_1_X128
    )
)
```

Once a position's fee growth has overflowed, any attempt to manage this position with the functions using the code pattern mentioned above will lead to a revert since the position's fee growth is now larger than the global fee growth. We observed that it is possible in some cases to have the position's fee growth re-overflow back into a range, which will allow these functions to work correctly again, but since this depends on tick location and specific timing, it's not a reliable method for users to recover their positions. This issue was caused by a translation error when rewriting from Solidity to Cairo, as the source implementation uses Solidity version `0.7.6`, which allows overflows on all integer arithmetic, while Cairo integers revert on overflows. In some areas of the code, this had been handled by the JediSwap team by specifically using functions like `mod_subtraction` to emulate an overflow, but this subtraction in the code pattern for the periphery contract was missed, causing transactions to revert.

Recommendation(s): Change the subtraction in the code pattern for `collect`, `increase_liquidity`, and `decrease_liquidity` to use an overflowing subtraction, as shown below:

```
mul_div(
    mod_subtraction(fee_growth_inside_0_X128, position_info.fee_growth_inside_0_last_X128),
    position_info.liquidity.into(),
    Q128
)
```

Status: Fixed

Update from the client: Fixed

7.3 [High] Swaps can fail due to overflow in new price calculations

File(s): sqrt_price_math.cairo

Description: When executing a swap, the function `get_next_sqrt_price_from_amount0_rounding_up` is called to calculate the new price. The formula used to calculate the new price after some amount of `token0` has been added/removed is `liquidity * sqrtPX96 / (liquidity ± amount * sqrtPX96)`. It is possible during this calculation for `liquidity * sqrtPX96` to be greater than what can be stored in a `u256` type, leading to an overflow in the Solidity source implementation. This overflow is intended behavior, with specific logic to check and handle an overflow if it occurs. The JediSwap implementation does this multiplication using the built-in `u256` type, which reverts on overflow, meaning in some cases during a swap execution, it will revert instead of overflowing and being handled properly. The function is shown below:

```
fn get_next_sqrt_price_from_amount0_rounding_up(
    sqrt_p_x96: u256, liquidity: u128, amount: u256, add: bool
) -> u256 {
    if (amount == 0) {
        return sqrt_p_x96;
    }

    let numerator = liquidity.into().shl(R96);
    // The overflow can occur here, leading to revert on swap
    let product = amount * sqrt_p_x96;

    if (add) {
        // This if statement checks if an overflow has occurred
        if (product / amount == sqrt_p_x96) {
            let denominator = numerator + product;
            if (denominator >= numerator) {
                return mul_div_rounding_up(numerator, sqrt_p_x96, denominator);
            }
        }
        return div_rounding_up(numerator, (numerator / sqrt_p_x96) + amount);
    } else {
        assert(product / amount == sqrt_p_x96, 'product overflows');
        assert(numerator > product, 'denominator negative');
        let denominator = numerator - product;
        let to_return = mul_div_rounding_up(numerator, sqrt_p_x96, denominator);
        assert(to_return <= MAX_UINT160, 'does not fit uint160');
        return to_return;
    }
}
```

Recommendation(s): Consider changing the multiplication of `amount * sqrt_p_x96` to allow for overflow in order to prevent transactions from reverting on certain swaps.

Status: Fixed

Update from the client: Fixed

7.4 [Info] Bitshift operation `shr` can return off-by-one incorrect value for negative `i256` inputs

File(s): `libraries/bitshift_trait.cairo`

Description: The function `shr(...)` for the type `i256` is designed to handle bitshift both when the input is positive or negative. We present the function below:

```
fn shr(self: @i256, n: i256) -> i256 {
    let mut new_mag = self.mag.shr(n.mag);
    let mut new_sign = *self.sign;
    if *self.sign && n.mag == 128 {
        new_mag += 1_u256;
    };
    if new_mag == 0 {
        if *self.sign {
            new_sign = true;
            new_mag = 1;
        } else {
            new_sign == false;
        };
    };
    IntegerTrait::<i256>::new(new_mag, new_sign)
}
```

This function should perform an arithmetic right shift operation on a two's complement representation of the signed number.

However, for certain inputs, the function does not return expected values. Below, we present a comparison of the outputs of right shifting a negative integer `-3` in Cairo and Solidity:

```
// Cairo implementation
let negative_three = IntegerTrait::<i256>::new(3, true);
let result = negative_three.shr(1);
result == IntegerTrait::<i256>::new(1, true);
// Result is -1
```

```
//Solidity implementation
int256 result = int256(-3 << 1);
result == -2;
// Result is -2
```

As shown above, the results of right shift operation differ. The Cairo implementation tries to achieve the arithmetic right shift of a two's complement number by dividing the number by 2^n. For example, `-3 » 1` is done by `-3 / 2`^{1}.

This approach returns correct values when the input is positive, but for negative inputs, an incorrect value may be returned. When `new_mag` is calculated, the magnitude of the input value is divided, rounding down. For a positive input, this is correct, but for a negative value, the magnitude is still divided and rounded down where it should actually be rounded up.

```
// -3 >> 1 in Cairo:
self.mag / 2^n
// The result is 1.5, but integer division will round down to 1
// On a negative integer this should have been rounded up from 1.5 to 2
3 / 2^1
3 / 2
1
```

Note that this issue has been addressed, the following statement is no longer necessary and should be removed to ensure that `shr` behaves correctly:

```
if *self.sign && n.mag == 128 {
    new_mag += 1_u256;
};
```

Recommendation(s): The described issue may be fixed by implementing the following changes:

- Rounding up the result of `mag` division, only if the input number is negative ;
- Removing the statement that adds one if the input number is negative and shift is `128` ;

Status: Fixed

7.5 [Info] Incorrect output for left bit shift by 128 for negative numbers

File(s): libraries/bitshift_trait.cairo

Description: The function `shl(...)` is presented below:

```
fn shl(self: @i256, n: i256) -> i256 {
    let mut new_mag = self.mag.shl(n.mag);
    if *self.sign && n.mag == 128 {
        new_mag += 1_u256;
    };

    if *self.sign {
        new_mag = new_mag & BoundedInt::<u256>::max() / 2;
    } else {
        new_mag = new_mag & ((BoundedInt::<u256>::max() / 2) - 1);
    };

    IntegerTrait::<i256>::new(new_mag, *self.sign)
}
```

If the input is negative and the number of bits to shift is exactly `128`, then `1` will be added to the absolute value of the number. Below we present an example of shifting the value `-1` by `128` bits:

```
// Cairo implementation
let minus_one = IntegerTrait::<i256>::new(1, true);
let result = minus_one.shl(128);
result == IntegerTrait::<i256>::new(340282366920938463463374607431768211457, true);
```

```
//Solidity implementation
int256 result = int256(-1 << 128);
result == -340282366920938463463374607431768211456
```

As shown in the example, the final results differ by `1` between Cairo and Solidity implementations. This change by one in `shl(...)` is not correct and should be removed to maintain the correct behavior.

Recommendation(s): Consider removing the statement that adds one to `new_mag` if the input number is negative and the shift is `128`.

Status: Fixed

7.6 [Info] Solidity signed integers don't support `abs`

File(s): libraries/signed_integers/*

Description: The signed integer types `i16`, `i32`, `i128`, `i256` implement the function `abs(...)` which return the absolute value of a given signed integer. Below, we present an example of this function for the `i32` type:

```
// Computes the absolute value of the given i32 integer.
fn i32_abs(a: i32) -> i32 {
    return i32 { mag: a.mag, sign: false };
}
```

Continuing with the `i32` type as an example, the valid ranges for positive and negative are shown below:

```
Positive:
            [0, 2^31-1]
            [0, 2147483647]
Negative:
            [-(2^31), -1]
            [-2147483648, -1]
```

If the `i32_abs` function is used on the minimum `i32` value of `-2147483648`, the magnitude will be kept the same, only unsetting the `sign` bit, resulting in a value of `2147483648`, which is one greater than the expected max range of `2147483647`.

It's not possible to support an `abs` function that returns its own type for this reason. The Solidity implementation doesn't have support `abs`, leaving developers to write it themselves if needed, for example, with the Uniswap-v3-core `abs` approach:

```
// `tick` is type int24
uint256 absTick = tick < 0 ? uint256(-int256(tick)) : uint256(int256(tick));
require(absTick <= uint256(MAX_TICK), 'T');
```

Note that in the code snippet above, the resulting absolute calculation is stored in a type larger than the original value to prevent this edge case on the signed integer minimum value.

Recommendation: There are two approaches to consider. The `abs` functionality can be kept by instead returning the unsigned equivalent for their type. This would mean that `i32_abs` would return a `u32` and so on. An alternative is to remove the `abs` functionality entirely and instead implement the abs logic outside of the library in a similar way to the uniswap-v3-core example.

Status: Fixed

Update from the client: Changed. Removed `abs`.

7.7 [Info] Missing overflow functionality on protocol and position fees

File(s): `jediswap_v2_pool.cairo`, `position.cairo`

Description: As part of the source implementation in Solidity, protocol fees are stored in a 128-bit unsigned integer. These protocol fees are able to overflow by design, so the protocol owner should claim their fees before this happens under normal circumstances. However, the translation from Solidity to Cairo did not account for this possible overflow. This makes it possible for the fees to exceed the size of a `u128`, and while it would normally have overflowed, it will now revert instead. The code is shown below:

```
fn swap(...) -> (i256, i256) {
    // ...
    // update fee growth global and, if necessary, protocol fees
    // overflow is acceptable, protocol has to withdraw before it hits type(uint128).max fees
    // @audit Incorrect comment, overflow causes revert
    if (zero_for_one) {
        self.fee_growth_global_0_X128.write(state.fee_growth_global_X128);
        if (state.protocol_fee > 0) {
            let mut protocol_fees = self.protocol_fees.read();
            protocol_fees.token0 += state.protocol_fee;
            self.protocol_fees.write(protocol_fees);
        }
    } else {
        self.fee_growth_global_1_X128.write(state.fee_growth_global_X128);
        if (state.protocol_fee > 0) {
            let mut protocol_fees = self.protocol_fees.read();
            protocol_fees.token1 += state.protocol_fee;
            self.protocol_fees.write(protocol_fees);
        }
    }
    // ...
}
```

It appears that the comment has been brought over from the source implementation, but the behavior of the translated code does not match what the comment states. A similar case exists in `PositionComponent::update` where the `position.tokens_owed0,1` values are updated during a position update:

```
fn update(...) {
    //...
    // overflow is acceptable, have to withdraw before you hit Q128 fees
    // @audit Incorrect comment, overflow causes revert
    position_info.tokens_owed_0 += tokens_owed_0;
    position_info.tokens_owed_1 += tokens_owed_1;
    //...
}
```

Reaching the maximum capacity of a `u128` in the examples shown above may lead to execution in `swap`, `mint`, and `burn` reverting unexpectedly. It should be noted that these revert cases can be easily resolved by claiming the protocol fees as the owner for `swap` and claiming position fees as a user for `mint/burn`. Additionally, reaching the maximum capacity of a `u128` without realizing fees is unlikely.

Recommendation(s): Consider introducing an internal function for overflowing addition to ensure that unexpected reverts cannot happen, or alternatively, the existing behavior can be left as is, and the incorrect comments can be removed.

Status: Fixed

Update from the client: Removed the comment.

7.8 [Info] Right bitshifts will revert instead of truncating significant bits

File(s): libraries/bitshift_trait.cairo

Description: For all supported types, including signed integers, the `U256BitShift` impl is used internally to calculate the shift. This shift implementation uses multiplication and division to calculate the result of a shift, as shown below:

```
impl U256BitShift of BitShiftTrait<u256> {
    #[inline(always)]
    fn shl(self: @u256, n: u256) -> u256 {
        *self * pow(2, n)
    }

    #[inline(always)]
    fn shr(self: @u256, n: u256) -> u256 {
        *self / pow(2, n)
    }
}
```

When shifting right, the result is `value * 2^n`, which results in a value that is the equivalent of a right bitshift. However, since this multiplication is done using the built-in `u256` type, if there were to be a shift that would lead to the significant bits being truncated, the internal `u256` library will calculate the result of the multiplication as being outside the valid range of a `u256`, and the transaction will revert. An example is shown below:

```
// Solidity

// hex(val) = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
val = 115792089237316195423570985008687907853269984665640564039457584007913129639935
val = val << 8
// 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00
val == 115792089237316195423570985008687907853269984665640564039457584007913129639680
```

```
// JediSwap

// hex(val) = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
val = 115792089237316195423570985008687907853269984665640564039457584007913129639935
val = val * pow(2,8)
    = val * 256
    = 29642774844752946028434172162224104410437116074403984394101141506025761187823360
// Exceeds the bounds of a u256 during mul, reverts
```

A similar issue exists for the bitshift implementations for `u32` and `u8`, where the result from the `pow` function can be higher than the supported type when attempting to unwrap, causing an `Option::unwrap_failed` error. A relevant code snippet is shown below:

```
impl U8BitShift of BitShiftTrait<u8> {
    #[inline(always)]
    fn shl(self: @u8, n: u8) -> u8 {
        *self * pow(2, n.into()).try_into().unwrap()
    }

    #[inline(always)]
    fn shr(self: @u8, n: u8) -> u8 {
        *self / pow(2, n.into()).try_into().unwrap()
    }
}
```

Recommendation(s): For the `u256` bitshift function, consider truncating `value` before the multiplication with `pow(2,n)` to prevent overflows on large shifts. For the smaller types, consider type casting the values to `u256` and using the standard function, then bitmask the result to truncate excess values before casting back to the intended type.

Status: Acknowledged

Update from the client: Acknowledged, never reaches such condition in the code.

7.9 [Info] Shifts where `n` is greater than `255` lead to revert

File(s): libraries/bitshift_trait.cairo

Description: The `BitShiftTrait` calculates the output of a bitshift by multiplying or dividing some target value by `pow(2,n)`. This is necessary because Cairo does not have native support for bitshifts at the time of this report. The `pow` function is shown below:

```
// Raise a number to a power.
fn pow(base: u256, exp: u256) -> u256 {
    if exp == 0 {
        1
    } else if exp == 1 {
        base
    } else if (exp & 1) == 1 {
        base * pow(base * base, exp / 2)
    } else {
        pow(base * base, exp / 2)
    }
}
```

The `pow` function doesn't have any protections against inputs that would result in a value that exceeds the maximum supported size of a `u256`. For these inputs, the function will revert. In isolation as a `pow` function, this behavior is reasonable, but when used for bitshift operations, the expected behavior when shifting would be to allow these large shifts and simply return zero instead of reverting. An example is shown below:

```
// Solidity

uint256 val = 0x1;
val = val << 256;
// The bit is pushed out entirely and truncated
val == 0
```

```
// Cairo

let mut val: u256 = 1;
// This will revert since 2^256 is greater than the max supported size of a `u256`
val = val.shl(256);
```

Recommendation(s): Consider checking the size of the shift, and if it exceeds what could be calculated by the `shl`, then return zero as all data will have been shifted and truncated.

Status: Acknowledged

Update from the client: Acknowledged, never reaches this condition.

7.10 [Info] Unused `owner` field in `JediSwapV2Factory`

File(s): jediswap_v2_factory.cairo

Description: The `JediSwapV2Factory` contract implements the OpenZeppelin `OwnableComponent` and uses its function `assert_only_owner` for access control. However, the contract's storage still includes a redundant `owner` field, which is not used and can be removed.

Recommendation(s): Consider removing the unused `owner` field from the `JediSwapV2Factory` storage.

Status: Fixed

Update from the client: Removed redundant owner.

7.11 [Best Practices] Factory upgrade function handles two classhashes

File(s): jediswap_v2_factory.cairo

Description: The JediSwapV2Factory function upgrade supports both upgrading the factory class hash and pool class hash in one call, as shown below:

```
fn upgrade(
    ref self: ContractState, new_class_hash: ClassHash, new_pool_class_hash: ClassHash
) {
    self.ownable_storage.assert_only_owner();
    if (!new_pool_class_hash.is_zero()
        && self.pool_class_hash.read() != new_pool_class_hash) {
        self.pool_class_hash.write(new_pool_class_hash);
        self.emit(UpgradedPoolClassHash { class_hash: new_class_hash });
    }
    self.upgradeable_storage._upgrade(new_class_hash);
}
```

The function logic is written such that upgrading the pool_class_hash is optional, but the factory class hash must be upgraded each time. If the protocol owners need to upgrade the pool_class_hash only, they will need to effectively set the factory class hash to the same hash, which is unnecessary and leads to an event being emitted stating that the class hash has been upgraded when it actually hasn't.

Recommendation(s): Consider splitting the functionality of upgrade into two separate functions for the factory class hash and the pool class hash.

Status: Acknowledged

7.12 [Best Practices] Simplify input for bitshift functions

File(s): libraries/bitshift_trait.cairo

Description: The bitshift trait specifies two functions shl and shr which accept arguments self and n, as shown below:

```
trait BitShiftTrait<T> {
    fn shl(self: @T, n: T) -> T;
    fn shr(self: @T, n: T) -> T;
}
```

The argument n specifies how many bits should be shifted, which can be represented with a type like usize rather than having n always match the type of self. In signed integer implementations, using the same type also increases complexity, having to deal with both the magnitude and sign.

Recommendation(s): Consider using a more approprate type for the argument n like usize which is specifically used for indexing and positioning.

Status: Acknowledged

8 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the JediSwap Protocol documentation

Throughout the review process, the JediSwap team offered assistance during meetings to clarify and address any questions or concerns raised by the Nethermind Security team. The reviewed code has NatSpec documentation and inline comments explaining the implementation details.

9 Test Suite Evaluation

9.1 Contracts Compilation

```
Updating git repository https://github.com/openzeppelin/cairo-contracts
    Updating git repository https://github.com/lambdaclass/yet-another-swap
   Compiling lib(jediswap_v2_core) jediswap_v2_core v0.0.1
    ↪  (/Users/admin/nethermind-audits/NM-0180-Jediswap/JediSwap-v2-core/Scarb.toml)
   Compiling starknet-contract(jediswap_v2_core) jediswap_v2_core v0.0.1
    ↪  (/Users/admin/nethermind-audits/NM-0180-Jediswap/JediSwap-v2-core/Scarb.toml)
    Finished release target(s) in 14 seconds
```

9.2 Tests Output

```
Tests: 482 passed, 8 failed, 0 skipped, 0 ignored, 0 filtered out

Failures:
    tests::test_libraries::test_sqrt_price_math::test_get_next_sqrt_price_from_input_returns_the_minumum_price
    _for_max_inputs
    tests::test_libraries::test_sqrt_price_math::test_get_next_sqrt_price_from_input_can_return_1_with_enough_
    amount_and_zero_for_one_equals_true
    tests::test_pool_swaps::test_swap_500_fee_1_1_price_large_liquidity_around_current_price_stable_swap_large_
    amount_zero_for_one_true_exact_input
    tests::test_pool_swaps::test_swap_500_fee_1_1_price_large_liquidity_around_current_price_stable_swap_large_
    amount_zero_for_one_true_exact_output
    tests::test_pool_swaps::test_swap_500_fee_1_1_price_large_liquidity_around_current_price_stable_swap_large_
    amount_zero_for_one_false_exact_input
    tests::test_pool_swaps::test_swap_500_fee_1_1_price_large_liquidity_around_current_price_stable_swap_large_
    amount_zero_for_one_false_exact_output
    tests::test_upgrade_factory::test_upgrade_succeeds_old_selector_fails
    tests::test_upgrade_pool::test_upgrade_succeeds_old_selector_fails
```

10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology 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 the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.