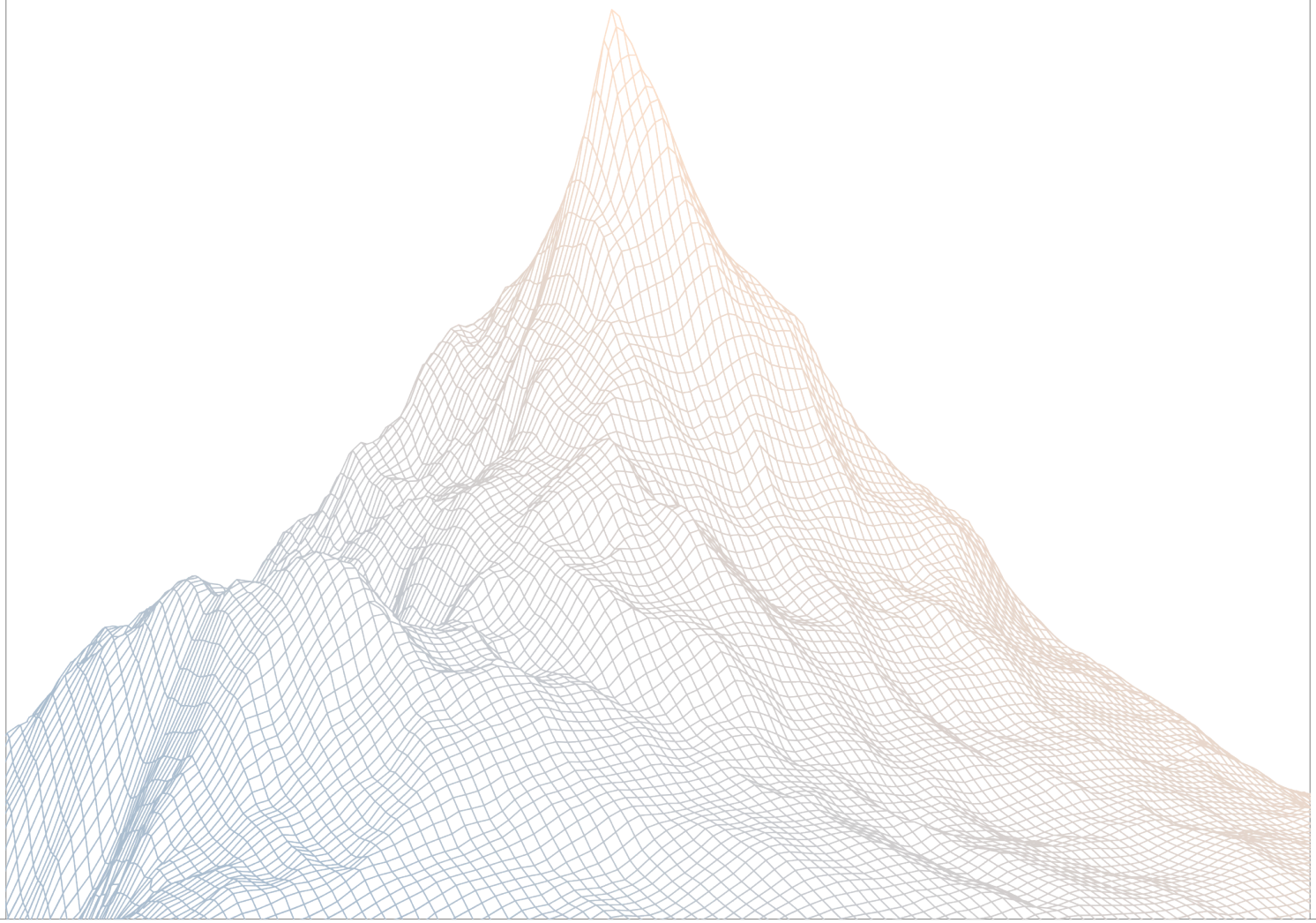
Zenith

Sushi

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: October 7th to October 31st, 2025

AUDITED BY: Bernd
Matte

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Sushi

Sushi, also known as SushiSwap, is the most multi-chain decentralized exchange (DEX) with the widest liquidity aggregation stack in DeFi. It operates across 40+ chains, aggregating liquidity sources across all these networks through its RouteProcessor 6 (RP6), ensuring users get the best swap prices across supported networks.

SushiSwap was created in late August 2020, offering various DeFi services as an automated market making (AMM), allowing users to swap tokens without the need for traditional market makers. It allows users to swap tokens and provide liquidity where they can stake their tokens to earn rewards. SushiSwap has its native token, $SUSHI, which gives holders governance rights and a share of the platform's transaction fees.

2.2 Scope

The engagement involved a review of the following targets:

Target	sushi-stellar
Repository	https://github.com/hyplabs/sushi-stellar.git
Commit Hash	`31cad6276ee3f2494bd2b0881a426659f502117d`

Files	`amm-math`
	`dex-factory`
	`dex-pool`
	`interfaces`

Target	sushi-stellar-periphery
Repository	https://github.com/hyplabs/sushi-stellar-periphery.git
Commit Hash	`1e6c05b00fd5ad8e32d6f08cb1622fe6c2b9b5f7`
Files	`contracts/**`

2.3 Audit Timeline

October 7, 2025	Audit start
October 31, 2025	Audit end
November 29, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	4
Medium Risk	7
Low Risk	8
Informational	11
Total Issues	**30**

3

Findings Summary

ID	Description	Status
H-1	Exact-output swaps reverse token directions, enabling pool drains	Resolved
H-2	Missing flash_callback authorization on flash loan callback allows bypassing integrator checks	Resolved
H-3	Incorrect liquidity position ownership model breaks NFT transfers and operator approvals	Resolved
H-4	Multiple NFTs can be minted for the same underlying pool LP position	Resolved
M-1	Oracle design leads to inaccurate price reporting	Resolved
M-2	Prefunded swaps pay out without consuming input, enabling pool siphoning	Resolved
M-3	Missing per-hop fees default to 3000, enabling unintended pool selection	Resolved
M-4	U256 overflow in Q96 arithmetic corrupts liquidity and amount calculations	Resolved
M-5	Using the NFT position manager's collect mechanism fails when using a different recipient than the NFT owner	Resolved
M-6	Oracle can return inaccurate prices by interpolating over large time windows	Resolved
M-7	Oracle's seconds-per-liquidity accumulator freezes when liquidity is zero, enabling TWAL manipulation	Resolved
L-1	Mint event misattributes sender, breaking off-chain attribution	Resolved
L-2	Protocol fee dust is permanently locked by "keep slot non-empty" logic	Resolved
L-3	Multi-hop exact input swap allows negative input amount which translates to exact output swap	Resolved
L-4	quote_exact_input_single does not resolve the price limit	Resolved
L-5	Potential LP state mismatch allows burning NFT	Resolved

ID	Description	Status
L-6	Fee growth calculation in NonfungiblePositionManager may panic due to unchecked subtraction	Resolved
L-7	Pool's reverse lookup key may expire	Resolved
L-8	Oracle price data has lower granularity than Sushi Swap V3 Solidity implementation	Resolved
I-1	Use overflow-safe oracle binary search calculation	Resolved
I-2	Outdated comment in Oracle's write function refers to previous tick instead of current tick	Resolved
I-3	Superfluous observe_single call in swap functions	Resolved
I-4	Redundant use of authorize_as_current_contract for subcontract calls	Resolved
I-5	Redundant pool unlock on error paths due to transaction atomicity	Resolved
I-6	Operator approval is not cleared on regular NFT transfer	Resolved
I-7	Oracle initialization does not store a recent observation	Resolved
I-8	Inconsistent operator approval mechanism and lack of approve_for_all support in the NonfungiblePositionManager contract	Resolved
I-9	Hardcoded value for observation TTL reduces readability	Resolved
I-10	Redundant TTL extensions in tick management	Resolved
I-11	Redundant explicit locking mechanisms	Resolved

4

Findings

4.1 High Risk

A total of 4 high risk findings were identified.

[H-1] Exact-output swaps reverse token directions, enabling pool drains

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- swap.rs

Description:

A sign/semantics error in `core_swap()` finalization reverses token directions for exact-output swaps (`amount_specified < 0`). During the swap loop, `update_swap_amounts()` sets `state.amount_calculated` to a positive value (the required input plus fee) and leaves `amount_specified - state.amount_specified_remaining` negative (the actual output delivered). At finalization, `core_swap()` unconditionally maps the pair as `(actual_input, actual_output)` for `zero_for_one` and `(actual_output, actual_input)` otherwise. This logic assumes exact-input but is also applied to exact-output, where the signs are inverted: for `zero_for_one` exact-output, it yields `amount0 < 0` (pool sends token0) and `amount1 > 0` (pool pulls token1), whereas the correct settlement is token0 in (positive), token1 out (negative).

Because transfers are keyed off sign, this inversion propagates into:

- `swap()`: `execute_swap_transfers()` sends negatives (pool→recipient) and pulls positives (sender→pool). Users can receive the wrong asset and send the wrong asset relative to intent, breaking settlement guarantees.
- `swap_prefunded()`: Designed to "skip incoming transfers and only perform outgoing transfers," it checks prefunding only on one side and `execute_swap_transfers_prefunded()` ignores positives entirely (it only sends negatives). With inverted signs in exact-output, the prefunding checks are bypassed and the pool sends out the wrong token without pulling the intended input, enabling a direct drain by any authorized router. No mode-aware guard corrects the mapping or forbids positives in the prefunded path.

```
fn poc_exact_output_direction_inversion_prefunded_drain() {
    // Demonstrates that exact-output mapping inverts directions for
    zero_for_one,
    // leading to the pool sending token0 (wrong) and not pulling token1 in.
    let ts = setup_diff();
    let TestSetup {
        env,
        wallet,
        pool,
        max_tick,
        min_tick,
        factory,
        token0,
        token1,
        pool_address,
        ..
    } = ts;
    env.cost_estimate().budget().reset_unlimited();

    // Add liquidity so pool holds tokens and can execute swaps
    let liquidity_amount = &(1e21 as u128);
    pool.mint(&wallet, &min_tick, &max_tick, &liquidity_amount);

    // Authorize a router for prefunded swaps
    let router = <soroban_sdk::Address as Address>::generate(&env);
    pool.set_router_authorized(&factory, &router, &true);

    // Prefund pool with token0 by doing a normal exact-input zero_for_one
    swap
    // This increases pool's token0 balance
    let _ = pool.swap(
        &wallet,
        &wallet,
        &true, // zero_for_one
        &(1e18 as i128), // exact-input
        &U256::from_u128(&env, MIN_SQRT_RATIO + 1),
    );

    // Record balances before the prefunded exact-output swap
    let pool0_before = token0.balance(&pool_address);
    let pool1_before = token1.balance(&pool_address);
    let router0_before = token0.balance(&router);
    let router1_before = token1.balance(&router);

    // Execute prefunded exact-output swap: amount_specified < 0
    // Due to direction inversion bug, pool will send token0 out (amount0 <
```

```
    0)
    // and not pull token1 in (amount1 > 0 ignored by prefunded transfer
    path)
    let res = pool
        .swap_prefunded(
            &router,
            &router,
            &true,                        // zero_for_one
            &(-(1e9 as i128)),           // exact-output request
            &U256::from_u128(&env, MIN_SQRT_RATIO + 1),
        );
    let res = res;

    // Assert signs reflect the inversion (amount0 negative means token0 sent
    out)
    assert!(res.amount0 < 0, "expected token0 to be sent (negative)");
    assert!(res.amount1 > 0, "expected token1 to be pulled (positive)");

    // Verify balances: router receives token0, router token1 unchanged
    let pool0_after = token0.balance(&pool_address);
    let pool1_after = token1.balance(&pool_address);
    let router0_after = token0.balance(&router);
    let router1_after = token1.balance(&router);

    assert!(router0_after > router0_before, "router should receive token0");
    assert_eq!(router1_after, router1_before, "router should not receive
    token1");

    // Pool loses token0, but pool token1 unchanged (no corresponding
    incoming transfer)
    assert!(pool0_after < pool0_before, "pool should lose token0 due to
    wrong-direction payout");
    assert_eq!(pool1_after, pool1_before, "pool should not gain token1 in
    prefunded path");
}
```

Recommendations:

- In `core_swap()`, map final `(amount0, amount1)` based on both `zero_for_one` and whether the swap is exact-input or exact-output, so that exact-output yields the correct sign orientation.
- In `swap_prefunded()`, treat positives as invalid (since the path promises no incoming transfers): either reject swaps where `amount0 > 0 || amount1 > 0` or verify prefunding on whichever side is positive and explicitly pull it (changing the semantics). For safety and simplicity, reject positives in prefunded mode.

- Add regression tests covering exact-output in both directions for both `swap()` and `swap_prefunded()` to assert correct signs and that the prefunded path never emits positives.

Minimal fix as a diff:

```
diff --git a/contracts/dex-pool/src/swap.rs b/contracts/dex-pool/src/swap.rs
-- a/contracts/dex-pool/src/swap.rs
++ b/contracts/dex-pool/src/swap.rs
@@ -186,6 +186,7 @@ fn core_swap(
     let exact_input = amount_specified > 0;

     let mut state = SwapState {
         ...
     };
@@ -315,10 +316,18 @@ fn core_swap(
     // Final amounts from tracked input/output
     let actual_input = amount_specified - state.amount_specified_remaining;
     let actual_output = state.amount_calculated;
    let (amount0, amount1) = if zero_for_one {
        (actual_input, actual_output)
    } else {
        (actual_output, actual_input)
    };
    let (amount0, amount1) = if zero_for_one {
        if exact_input {
            (actual_input, actual_output)
        } else {
            (actual_output, actual_input)
        }
    } else {
        if exact_input {
            (actual_output, actual_input)
        } else {
            (actual_input, actual_output)
        }
    };

     Ok(CoreSwapResult { ... })
@@ -446,19 +455,31 @@ pub fn swap_prefunded(
     let core = core_swap(&env, zero_for_one, amount_specified,
    &sqrt_price_limit_x96)?;
```

```
// Safety: ensure sufficient prefunded input exists in pool
let pool_addr = env.current_contract_address();
if zero_for_one {
    if core.amount0 > 0 {
        let bal0 = Token**Sushi:**:new(&env, &core.pool_state.params.
            token0).balance(&pool_addr);
        if bal0 < core.amount0 {
            return Err(Error::InsufficientToken0);
        }
    }
} else if core.amount1 > 0 {

    let bal1 = Token**Sushi:**:new(&env, &core.pool_state.params.token1).
        balance(&pool_addr);
    if bal1 < core.amount1 {
        return Err(Error::InsufficientToken1);
    }
}
// Prefunded mode performs no incoming transfers; reject any positive
    amounts
if core.amount0 > 0 || core.amount1 > 0 {
    return Err(Error::InvalidAmount);
}

// (Optional defense-in-depth) If you prefer to allow positives,
// validate prefunding for whichever side is positive:
// let pool_addr = env.current_contract_address();
// if core.amount0 > 0 {
//     let bal0 = Token**Sushi:**:new(&env, &core.pool_state.params.
    token0).balance(&pool_addr);

//     if bal0 < core.amount0 { return Err(Error::InsufficientToken0); }
// }
// if core.amount1 > 0 {
//     let bal1 = Token**Sushi:**:new(&env, &core.pool_state.params.
    token1).balance(&pool_addr);

//     if bal1 < core.amount1 { return Err(Error::InsufficientToken1); }
```

```
    // }

     // Perform only outgoing transfers
     execute_swap_transfers_prefunded(&env, &recipient, core.amount0,
    core.amount1, ...)
```

Sushi: Resolved with PR-193.

Zenith: Verified.

[H-2] Missing `flash_callback` authorization on flash loan callback allows bypassing integrator checks

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/flash.rs#L155-L160

Description:

The `flash` function allows initiating flash loans. It accepts a `callback_contract` address, which is invoked to execute logic in the `flash_callback` function with the loaned funds and handle repayment.

However, the implementation fails to check whether `callback_contract` is authorized, i.e., `callback_contract.require_auth()`. This means it does not check if the initiator of the `flash` call is the callback contract itself. This allows any arbitrary address to trigger a flash loan on behalf of `callback_contract` and specify any other `recipient` address to receive the loaned funds.

```
153: // 7. Execute callback
154: // The callback must transfer repayment back to the pool; we verify
    deltas below.
155:  let cb_args = (env.current_contract_address(), fee0, fee1,
    data.clone()).into_val(&env);
156:  env.invoke_contract::<()>(
157:      &callback_contract,
158:      &Symbol::new(&env, "flash_callback"),
159:      cb_args,
160:  );
```

This is problematic for contracts designed to integrate with the DEX pool's flash loan functionality. Such an integrator contract might implement critical pre-loan checks, such as access control for who can initiate a flash loan or validation of loan parameters, hence, only allowing the flash loan to be initiated by itself.

By calling the DEX's `flash` function directly, anyone can specify the integrator contract as the callback handler, bypassing these essential checks and potentially exploiting the integrator's logic.

Recommendations:

Consider calling `callback_contract.require_auth()` at the start of the `execute_flash` function to enforce that the authorization is provided.

Sushi: Resolved with PR-232. We are now using the begin and end pattern for the flash. You can get how the flash loan flow works from the FlashExecutor code comments at the start.

Zenith: Verified.

[H-3] Incorrect liquidity position ownership model breaks NFT transfers and operator approvals

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- nonfungible_position_manager.rs#L564

Description:

In the `NonfungiblePositionManager` contract, the `mint` function creates a liquidity position in the underlying pool for a user-specified `recipient` address. Subsequent interactions with this position, such as increasing or decreasing liquidity via `increase_liquidity` and `decrease_liquidity`, or collecting fees via `collect`, are performed on behalf of the current owner of the corresponding NFT with the corresponding authorizations. In addition, the underlying pool LP position can be managed directly by the original minter address via the pool contract.

This design deviates from the battle-tested model used in the Sushi Swap v3 Solidity implementation, where the NFT position manager contract itself is the owner of a pool LP position and all user NFT positions are aggregated into it. This discrepancy leads to issues.

Specifically, this implementation has two major impacts:

1. **Broken NFT position transfers**: If an NFT holder transfers their token to a new owner, the new owner is unable to manage the associated liquidity position. The underlying pool has the liquidity registered to the original NFT minter's address. When the new owner attempts to modify the position, the manager contract calls the pool using the new owner's address, which does not own the liquidity. This causes the transaction to fail, effectively making the NFT positions non-transferable.

2. **Incompatibility with the operator pattern**: The implementation prevents the delegation of position management to an operator. Functions that modify liquidity in the pool require authorization from the position owner. When an operator, approved by the NFT owner, calls a function on the `NonfungiblePositionManager`, the manager then calls the pool using the owner's address. The pool, in turn, requires authorization from the owner's address (`owner.require_auth()`). However, since the transaction was initiated by the operator, this authorization check fails, making it impossible for operators to manage positions on behalf of the owners.

Likely, this issue was not discovered during testing because the test suite uses `env.mock_all_auths()`, which bypasses all authorization checks.

Recommendations:

Consider aligning the contract's logic with the reference Solidity implementation. The `NonfungiblePositionManager` contract should be the direct owner of the liquidity positions in the underlying pools.

Note that in this model, it is crucial that the manager contract is designed to securely handle and segregate user funds, ensuring that each user's liquidity and fees are correctly tracked and managed within the contract itself.

Sushi: Resolved with PR-117.

Zenith: Verified.

[H-4] Multiple NFTs can be minted for the same underlying pool LP position

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- nonfungible_position_manager.rs#L547-L644

Description:

The `mint` function in the `NonfungiblePositionManager` contract creates a new liquidity position in the pool and issues an NFT to represent ownership. However, the function lacks a crucial check to verify if a position already exists for the same set of parameters (i.e., the same recipient, token pair, fee, and tick range).

When the NFT position manager's `mint` function is called, it invokes the underlying pool's `mint` function. If a liquidity position for the specified recipient and ticks already exists, the pool contract adds liquidity to that existing position. Despite this, the `NonfungiblePositionManager` proceeds to mint a new, distinct NFT for what is effectively the same underlying position.

This flaw allows the creation of multiple NFTs that all reference and control a single liquidity position. This breaks the core assumption of an NFT representing a unique, non-fungible position. A malicious user could exploit this by minting multiple NFTs that represent the same underlying pool LP position, selling one of it, and continuing to use the other NFT, which would grant them continued control over the entire position.

Recommendations:

Consider modifying the `mint` function to prevent the creation of more than one NFT for a single pool LP position.

Sushi: Resolved with PR-127.

Zenith: Verified.

4.2 Medium Risk

A total of 7 medium risk findings were identified.

[M-1] Oracle design leads to inaccurate price reporting

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/oracle.rs#L214-L222

Description:

The oracle's storage architecture is divided into two tiers:

- temporary storage for high-granularity (every 60 seconds) recent observations and
- persistent storage for low-granularity historical checkpoints, which are saved less frequently (hourly)

When a query is made for a time that falls within the historical observations, the observe function performs linear interpolation between two data points that are 1 hour distant. This long-period interpolation effectively averages out or "smears" any price volatility that occurred within the gap, leading to a Time-Weighted Average Price (TWAP) that may not accurately reflect the true market conditions at the queried time. This behavior diverges from the Sushi Swap V3 Solidity reference implementation, which uses a contiguous circular buffer to maintain consistent data granularity.

For example, an attacker could manipulate the price for a short duration during a period of low activity, causing the effect to be averaged over a much longer period. If an attacker shifts the price by 1000 ticks for 5 minutes (300 seconds) within a 1-hour (3600 seconds) interpolation gap, the average tick will be biased by approximately `1000 * (300 / 3600) ≈ 83.3` ticks. This translates to a lasting price bias of nearly 1% for any query made within that hour, which is sufficient to cause unfair liquidations or other economic exploits in dependent protocols.

Recommendations:

Consider one of the following approaches to mitigate this risk:

1. **Short-term mitigation**: Increase the frequency of persistent checkpoints by significantly reducing the CHECKPOINT_INTERVAL. This would decrease the maximum possible interpolation gap, thereby reducing the window for manipulation. However, this comes with increased transaction costs to extend/restore checkpoints.
2. **Long-term solution**: For a more robust solution that aligns with the security model of the original Sushi Swap V3 Solidity contracts, also consider using a fixed-size circular buffer with uniform sampling. This would prevent large data gaps from forming and ensure that TWAP calculations are always based on a high-granularity, contiguous set of recent price data.

Sushi: Resolved with PR-203. Just wanted to mention that we are allowing checkpoints to expire naturally with TTL as this already gives us better parity (Uniswap has max 9 days history of observations). So we will not need to extend or restore these checkpoints.

Zenith: Verified.

[M-2] Prefunded swaps pay out without consuming input, enabling pool siphoning

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- swap.rs

Description:

The prefunded swap path allows payouts from pool reserves without enforcing any corresponding input consumption. In `swap_prefunded()`, the code computes final amounts via `core_swap()` and performs a shallow "prefund" check against the pool's current balances before calling `execute_swap_transfers_prefunded()`. However:

- `swap_prefunded()` only checks the pool's global balance for whichever side is (assumed) positive, measuring all reserves (including LP liquidity and other funds) rather than per-swap attribution.
- `execute_swap_transfers_prefunded()` intentionally skips all positive amounts and only performs outgoing transfers for negatives, meaning no input is ever pulled in this mode.
- Therefore, an allowlisted router can invoke `swap_prefunded()` and receive the negative leg while contributing no input, as long as the pool currently holds sufficient reserves. This includes tokens left in the pool for any reason (e.g., protocol-fee accounting errors, dust, or any prefund from previous flows).

Highest impact scenario: An authorized router calls `swap_prefunded()` with a permissive price limit and extracts value directly from LP reserves (e.g., zero_for_one exact-input: pool sends out token1, does not pull token0). Price and fees update as if input were supplied, yet the input is not consumed in transfers. The allowlist (`AUTH_ROUTERS`) limits callers but does not prevent a malicious/compromised router from abusing this path.

Recommendations:

- If positives are allowed, then enforce actual consumption: either pull the positive leg from a designated prefund escrow or attribute per-call deposits that are consumed during the swap, and validate they match the required input.
- Add regression tests ensuring prefunded swaps never pay out unless input is provably consumed or both amounts are non-positive.

Minimal fix as a diff:

```
diff --git a/contracts/dex-pool/src/swap.rs b/contracts/dex-pool/src/swap.rs
-- a/contracts/dex-pool/src/swap.rs
++ b/contracts/dex-pool/src/swap.rs
@@ -443,6 +443,16 @@ pub fn swap_prefunded(
     // Compute core swap (locks pool and prepares updates)
     let core = core_swap(&env, zero_for_one, amount_specified,
    &sqrt_price_limit_x96)?;

    // Prefunded mode performs no incoming transfers; reject any positive
        amounts.
    // This ensures the path never relies on global pool balances as
        implicit input.
    if core.amount0 > 0 || core.amount1 > 0 {
        return Err(Error::InvalidAmount);
    }

    // (If you must allow positives, replace the early return above with
    // explicit attribution and pulling of the positive leg from a known
        escrow
    // or per-call prefund record before proceeding.)

     // Safety: ensure sufficient prefunded input exists in pool
     let pool_addr = env.current_contract_address();
     if zero_for_one {
```

Sushi: Resolved with PR-207.

Zenith: Verified.

[M-3] Missing per-hop fees default to 3000, enabling unintended pool selection

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- quote.rs
- multi_hop.rs

Description:

In multi-hop routing, each hop must be unambiguously defined by `(token_in, token_out, fee)`. The router currently substitutes a default fee tier of `3000` whenever a per-hop fee is missing. Specifically:

- In `quote_exact_input()`, the code derives the fee for hop `i` using `params.fees.get(i).unwrap_or(3000)`. There is no length validation on `params.fees`, so an underspecified `fees` array is silently accepted and the router selects the `3000`-fee pool.
- In `execute_exact_input()`, similar `unwrap_or(3000)` lookups exist; however, `swap_exact_input()` calls `validate_path()` before execution, which enforces `fees.len() == path.len() - 1`. This makes the execution path safe today, but the default remains in the code and is a latent footgun if validation is ever bypassed or reused elsewhere.

Root cause:

- Pool selection is security-critical (it defines which pool receives the trade). Defaulting a missing fee to `3000` allows the router to choose a pool the caller did not intend. This deviates from the v3 pattern on EVM (Uniswap/Sushi), where path encoding forces an explicit fee per hop and invalid paths revert.

```
use crate::swap_router_tests::support::{init_router_factory, MockFactory,
    MockFactoryClient, MockPool};
use crate::{types::ExactInputParams, SwapRouter, SwapRouterClient};
use soroban_sdk::testutils::Address as _;
use soroban_sdk::{vec, Address, Env};

#[test]
```

```
fn test_quote_defaults_fee_to_3000_misroutes_to_unintended_pool() {
    let env = Env::default();
    env.mock_all_auths();

    // Setup router and factory
    let factory = env.register(MockFactory, ());
    let router_id = env.register(SwapRouter, ());
    let client = SwapRouter**Sushi:**:new(&env, &router_id);
    init_router_factory(&env, &router_id, &factory);

    // Same pair with two pools: 500 and 3000
    let a = Address::generate(&env);
    let b = Address::generate(&env);
    let pool_500 = env.register(MockPool, ());
    let pool_3000 = env.register(MockPool, ());
    let factory_client = MockFactory**Sushi:**:new(&env, &factory);
    factory_client.set_pool(&a, &b, &500u32, &pool_500);
    factory_client.set_pool(&a, &b, &3000u32, &pool_3000);

    // Missing fees → quote defaults to 3000 and succeeds via fee=3000 pool
    let params_missing = ExactInputParams {
        sender: Address::generate(&env),
        path: vec![&env, a.clone(), b.clone()],
        fees: vec![&env], // missing per-hop fee
        recipient: Address::generate(&env),
        amount_in: 1000,
        amount_out_minimum: 1,
        deadline: 9_999_999_999,
    };
    let quoted = client.quote_exact_input(&params_missing);
    assert!(quoted.amount > 0); // proves defaulting enabled pool selection

    // If only a 500-fee pool exists, the same missing-fee quote fails (no
    3000 pool)
    let pool_only_500 = env.register(MockPool, ());
    let a2 = Address::generate(&env);
    let b2 = Address::generate(&env);
    factory_client.set_pool(&a2, &b2, &500u32, &pool_only_500);

    let params_missing_fail = ExactInputParams {
        sender: Address::generate(&env),
        path: vec![&env, a2.clone(), b2.clone()],
        fees: vec![&env], // still missing
        recipient: Address::generate(&env),
        amount_in: 1000,
        amount_out_minimum: 1,
        deadline: 9_999_999_999,
```

```
    };
    let res = client.try_quote_exact_input(&params_missing_fail);
    assert!(res.is_err()); // fails only because defaulted 3000 pool is
    absent
}
```

Recommendations:

```
-- a/contracts/swap-router/src/quote.rs
++ b/contracts/swap-router/src/quote.rs
@@
 use crate::errors::SwapRouterError;
 use crate::price_limits::get_sqrt_price_limit;
 use crate::swap::{get_pool_address, PoolClient};
use crate::types::{ExactInputParams, ExactOutputParams};
use crate::types::{ExactInputParams, ExactOutputParams};
use crate::validation::validate_path;
@@
 pub fn quote_exact_input(
     env: &Env,
     params: &ExactInputParams,
 ) -> Result<QuoteResult, SwapRouterError> {
    let path = &params.path;
    if path.len() < 2 {
        return Err(SwapRouterError::InvalidPath);
    }
    // Enforce the same path/fees rules as execution
    validate_path(&params.path, &params.fees)?;

    let mut amount_out = params.amount_in;
    let path = &params.path;
    let mut amount_out = params.amount_in;
     let mut last_price = U256::from_u128(env, 0);
@@
    for i in 0..path.len() - 1 {
    for i in 0..path.len() - 1 {
        let token_in = &path.get(i).unwrap();
        let token_out = &path.get(i + 1).unwrap();
        let fee = params.fees.get(i).unwrap_or(3000);
        let fee = *params
```

```
            .fees
            .get(i)
            .ok_or(SwapRouterError::InvalidPath)?;
@@
        let result =
            quote_exact_input_single(env, token_in, token_out, fee,
    amount_out, sqrt_limit)?;
```

Sushi: Resolved with PR-118.

Zenith: Verified.

[M-4] U256 overflow in Q96 arithmetic corrupts liquidity and amount calculations

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- liquidity_amounts.rs

Description:

The library performs fixed-point math on Q96-scaled `U256` values by multiplying two large operands and only then dividing (e.g., `a.mul(b).div(Q96)`, `(liquidity << 96).mul(b - a) / b / a`). On Soroban, `U256` operations are modulo 2^256; intermediate products that exceed 256 bits silently wrap (or cause arithmetic domain errors), producing incorrect results. This pattern appears in `get_liquidity_for_amount_0()`, `get_amount_0_for_liquidity()`, and `get_amount_1_for_liquidity()`.

Root cause:

- Multiplication of two Q96-scaled square-root prices (or large intermediates like `(liquidity << 96) * (b - a)`) can exceed 256 bits before division is applied.
- The implementation relies on plain `U256` multiplication followed by division, which does not preserve precision and can overflow for realistic tick bounds.
- Uniswap V3 avoids this class of bugs by using a 512-bit intermediate and performing a single `mulDiv` operation; this codebase already exposes an equivalent via `amm-math::full_math::mul_div()`.

Highest impact scenario:

- With ticks near the extremes and modest liquidity/amount inputs, the naive multiply-then-divide overflows, causing wrapped values and materially wrong liquidity and token amount calculations. This can break quotes, misprice liquidity provisioning/withdrawal, and lead to value loss or protocol-level accounting errors.

A proof-of-concept test demonstrates the issue by selecting ticks close to the maximum, where naive `(a*b)/Q96` diverges from `full_math::mul_div(a,b,Q96)` and triggers an overflow error.

```
use amm_math::{full_math, tick_math, try_u256_to_u128};
use soroban_sdk::{Env, U256};
```

```
#[test]
fn test_liquidity_amounts_overflow_poc() {
    let env = Env::default();
    let tick_low = 887_270;  // near MAX_TICK
    let tick_high = 887_272; // MAX_TICK

    let a = tick_math::get_sqrt_price_at_tick(&env, tick_low).unwrap();
    let b = tick_math::get_sqrt_price_at_tick(&env, tick_high).unwrap();

    let q96 = U256::from_u128(&env, 1u128 << 96);
    let naive = a.mul(&b).div(&q96); // may overflow
    let safe = full_math::mul_div(&env, &a, &b, &q96).unwrap(); // 512-bit
    intermediate
    assert!(naive ≠ safe);

    let amount0 = U256::from_u128(&env, 1_000_000);
    let b_minus_a = b.sub(&a);
    let safe_liq = full_math::mul_div(&env, &amount0, &safe,
    &b_minus_a).unwrap();
    let naive_liq = amount0.mul(&naive).div(&b_minus_a);

    let _ = try_u256_to_u128(&env, &safe_liq).unwrap();
    let _ = try_u256_to_u128(&env, &naive_liq).unwrap_or(u128::MAX);

    assert!(safe_liq ≠ naive_liq);
}
```

Recommendations:

Replace all "multiply then divide" sequences on `U256` with `full_math::mul_div()` (or `mul_div_rounding_up()` where rounding-up is required). Where multiple divisions are chained, restructure to keep intermediates bounded or use sequential `mul_div` steps that never reintroduce a risky wide numerator.

Example patch for `liquidity_amounts.rs` (directional):

```
-- a/contracts/libraries/src/liquidity_amounts.rs
++ b/contracts/libraries/src/liquidity_amounts.rs

 use amm_math::{try_u256_to_u128, SqrtPriceX96};
use amm_math::full_math;

        let intermediate = a.mul(b).div(&U256::from_u128(env, Q96));
```

```
        // Use 512-bit intermediate to avoid overflow: floor(a * b / Q96)
        let intermediate = full_math::mul_div(
            env,
            a,
            b,
            &U256::from_u128(env, Q96),
        )
        .unwrap();


    try_u256_to_u128(env, &amount0_u256.mul(&intermediate).div(&b.sub(a))
            ).unwrap()

    // amount0 * intermediate / (b - a) using mul_div to keep intermediate
            s safe
        let liq = full_math::mul_div(env, &amount0_u256, &intermediate, &b.
            sub(a)).unwrap();
        try_u256_to_u128(env, &liq).unwrap()

        try_u256_to_u128(
            env,
            &amount1_u256.mul(&U256::from_u128(env, Q96)).div(&b.sub(a)),
        )
        .unwrap()
        let num = full_math::mul_div(env, &amount1_u256, &U256::from_u128(
            env, Q96), &b.sub(a))
            .unwrap();
        try_u256_to_u128(env, &num).unwrap()

        let amount0_u256 = U256::from_u128(env, liquidity)
            .shl(Q96_RESOLUTION)
            .mul(&b.sub(a))
            .div(b)
            .div(a);
        // ((liquidity << 96) * (b - a)) / b / a, done in safe steps
        let liq_q96 = U256::from_u128(env, liquidity).shl(Q96_RESOLUTION);

    let step1 = full_math::mul_div(env, &liq_q96, &b.sub(a), b).unwrap();
        let amount0_u256 = step1.div(a);
```

```
let amount1_u256 = U256::from_u128(env, liquidity)
    .mul(&b.sub(a))
    .div(&U256::from_u128(env, Q96));
let amount1_u256 = full_math::mul_div(
    env,
    &U256::from_u128(env, liquidity),
    &b.sub(a),
    &U256::from_u128(env, Q96),
)
.unwrap();
```

Sushi: Resolved with @cd8a1c9447.. .

Zenith:

[M-5] Using the NFT position manager's `collect` mechanism fails when using a different recipient than the NFT owner

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- nonfungible_position_manager.rs#L897

Description:

In the `NonFungiblePositionManager`, the `collect` function is responsible for collecting funds for a given liquidity position, identified by a `token_id`. It accepts a `recipient` address in its `CollectParams`, which is then passed to the underlying pool's `collect` function.

```
888: // Call pool.collect() which will:
889: // 1. Update position fees internally (no need for burn(0))
890: // 2. Cap collection at pool's tokens_owed
891: // 3. Transfer tokens and return actual amounts
892: let collect_result: (u128, u128) = env.invoke_contract(
893:     &pool,
894:     &symbol_short!("collect"),
895:     vec![
896:         &env,
897:         recipient.into_val(&env),
898:         position.tick_lower.into_val(&env),
899:         position.tick_upper.into_val(&env),
900:         params.amount0_max.into_val(&env),
901:         params.amount1_max.into_val(&env),
902:     ],
903: );
```

However, the pool contract identifies liquidity positions by the owner's address. As a result, when using an arbitrary `recipient` address, the function will fail to find the correct position and errors. This means it is not possible to send collected funds to an arbitrary address.

Recommendations:

First, consider adopting a similar approach to the Solidity implementation, using the NFT position manager as the pool LP owner (as reported in another finding). This allows using the position manager's address as the recipient when calling `collect`, followed by transferring the collected funds to the desired recipient.

Sushi: Resolved with PR-204. The Core now accepts a sender parameter in the collect function. the particular change to include the sender parameter was done in PR-111 in the Periphery Repo, but the change in the periphery Repo where we need to have the current contract address as the liquidity position owner as well for the collect function was done in PR-117.

Zenith: Verified.

[M-6] Oracle can return inaccurate prices by interpolating over large time windows

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/oracle.rs#L563

Description:

The oracle's `find_surrounding_observations` function is designed to find two observations that bracket a target timestamp, which are then used for interpolation.

The function first performs a binary search on a set of recent observations stored in temporary storage. If no suitable observations are found there (for instance, if they have all expired due to their shorter TTL), the function falls back to searching through historical checkpoints stored in persistent storage.

However, this can become problematic if all recent observations have expired, but the `LatestOracleObservation` (stored in instance storage with a longer TTL) is still available. In this scenario, the oracle might find an old `before_obs` from the historical checkpoints but will fail to find a corresponding `after_obs`. The logic in line 563 then defaults to using the latest observation as the `after_obs`.

```
561: // Ensure we found both observations
562: let before = before_obs.ok_or(Error::InvalidObservation)?;
563: let after = after_obs.unwrap_or(latest); // Use latest if no after found
```

This can create a significant time gap between the before and after observations, forcing interpolation over a potentially very long and outdated period. Such a large window increases the risk of price manipulation, as the interpolated price will not accurately reflect the market conditions between the two distant points. This behavior deviates from implementations like the Sushi Swap v3 Solidity implementation, uses a contiguous circular buffer to maintain consistent data granularity and errors on out-of-bounds queries.

Recommendations:

Consider implementing a stricter policy for observation lookups. Instead of defaulting to the latest observation when a proximate `after` observation is not found, the function should revert.

Alternatively, enforce a maximum allowable time gap between the `before` and `after` observations used for interpolation. If the gap exceeds this threshold, the transaction should fail, preventing the calculation of a potentially manipulable and inaccurate price.

Sushi: Resolved with PR-201. We tightened the oracle to fail fast instead of interpolating across stale data. Specifically, we added a hard limit on the time span between the two observations used for interpolation. If that span is larger than our checkpoint cadence (now ~30 minutes), we return an explicit ObservationTooOld error rather than producing an unreliable value. We also halved the checkpoint interval from ~1 hour to ~30 minutes. This should reduce the worst-case gap and makes valid interpolation more likely, even when a pool is quiet.

Zenith: Verified.

[M-7] Oracle's seconds-per-liquidity accumulator freezes when liquidity is zero, enabling TWAL manipulation

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/oracle.rs#L428-L431

Description:

The oracle's `seconds_per_liq_cum_x128` accumulator is designed to provide a time-weighted measure of liquidity, which is critical for downstream protocols to assess pool health and risk. The current Soroban implementation deviates from the canonical Sushi Swap V3 behavior when a pool's liquidity drops to zero. Instead of recording this event as a period of maximum illiquidity (infinite cost), the accumulator freezes, effectively erasing the market failure from the historical record.

This behavior is located in `contracts/dex-pool/src/oracle.rs`, where the accumulation logic explicitly returns a zero delta if `liquidity = 0`. This is a direct contradiction to the reference Sushi Swap V3 implementation, which uses `max(1, liquidity)` as the denominator to ensure the accumulator continues to grow at its maximum possible rate during a liquidity outage.

As a consequence, the oracle treats a state of infinite cost (zero liquidity) as having zero cost. This allows manipulation of the Time-Weighted Average Liquidity (TWAL). If a pool's liquidity is shortly 0, the pool appears more liquid and stable than it actually was. Any protocol relying on this oracle for functions like determining collateral value or safe trade sizes could be misled into making unsafe, exploitable decisions.

To demonstrate the vulnerability, we analyze a 15-second time window (`□t = 15s`) and calculate the TWAL. The formula is `TWAL = (□t * 2^128) / □S_x128`, where `□S_x128` is the change in the accumulator over the period.

Scenario A: Normal Operation (Liquidity is never zero)

In this scenario, liquidity fluctuates but remains positive. Both the Soroban implementation and the Sushi Swap V3 Solidity logic produce the same correct result.

- **Interval 1 (5s):** Liquidity = `1.0 x 10^9`. `delta` ≈ `1.701 x 10^30`
- **Interval 2 (5s):** Liquidity = `0.8 x 10^9`. `delta` ≈ `2.127 x 10^30`
- **Interval 3 (5s):** Liquidity = `1.2 x 10^9`. `delta` ≈ `1.418 x 10^30`

- **Total Accumulator Change (□*S_x128*):** `5.246 x 10^30`
- **Calculated TWAL:** `(15 * 2^128) / (5.246 x 10^30)` ≈ **972,931,757** ≈ 973 million

This result is a sensible and accurate time-weighted average of the pool's liquidity.

Scenario B: Market Failure (Brief Zero-Liquidity)

Here, liquidity is 0 for 5 seconds. The two implementations now produce different results.

- **Interval 1 (5s):** Liquidity = `1.0 x 10^9`. Both add `~1.701 x 10^30`.
- **Interval 2 (5s):** Liquidity = **0**.
  - **Soroban Logic:** `delta` = **0** (Accumulation freezes).
  - **Uniswap V3 Solidity Logic:** `delta` = `(5 * 2^128) / 1` ≈ *1.701 x 10^39* (Maximum penalty applied).
- **Interval 3 (5s):** Liquidity = `1.0 x 10^9`. Both add `~1.701 x 10^30`.

Results Comparison:

Implementation	Total Accumulator Change (`ΔS_x128`)	Calculated TWAL	Logical Interpretation
Soroban Logic	`3.402 x 10^30`	**1,500,000,000**	Incorrectly reports the pool was **50% more liquid** than normal, ignoring the failure.
Uniswap V3 Logic	`1.701 x 10^39`	**3**	Correctly signals a critical failure, reporting an extremely low average liquidity.

Recommendations::

To ensure the oracle is manipulation-resistant and compatible with the broader DeFi ecosystem, it is critical to align its behavior with the Solidity Sushi Swap V3 implementation.Modify the `calculate_seconds_per_liquidity_delta` function to treat zero liquidity as a denominator of one. This correctly applies a penalty for periods of market failure.

Additionally, the conditional logic that freezes the accumulator in `write` and `observe_single` should be removed, allowing the calculated `delta` to be applied unconditionally.

Sushi: Resolved with PR-202. We corrected the zero-liquidity behavior to match Uniswap V3. Instead of freezing the seconds-per-liquidity accumulator when liquidity is zero, we now treat zero as one (max penalty) in the denominator. So any period with zero liquidity drives the accumulator up at the maximum rate, which translates to a very low TWAL for

that window.

Zenith: Verified.

4.3 Low Risk

A total of 8 low risk findings were identified.

[L-1] Mint event misattributes sender, breaking off-chain attribution

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- lib.rs

Description:

The `MintEvent` emitted by `Pool::mint()` records `sender` as the pool's own address (`env.current_contract_address()`) rather than the actual minting actor (the LP or invoker). This diverges from Uniswap V3 semantics where `Mint` events attribute `sender` to the external caller, and causes incorrect attribution for indexers, risk monitors, and analytics pipelines that rely on this field to identify who initiated the mint.

Recommendations:

Log the actual actor as `sender` for `MintEvent`. Given `Pool::mint()` enforces `recipient.require_auth()`, using `recipient` provides clear, authenticated attribution. Alternatively, use `env.invoker()` if you prefer the direct caller semantics. Keep event fields consistent with Uniswap V3 to preserve ecosystem tooling compatibility.

```
diff --git a/contracts/dex-pool/src/lib.rs b/contracts/dex-pool/src/lib.rs
-- a/contracts/dex-pool/src/lib.rs
++ b/contracts/dex-pool/src/lib.rs
@@ -996,7 +996,7 @@ impl Pool {
         final_pool_state.slot0.unlocked = true;
         Self::set_pool_state(&env, &final_pool_state);

         MintEvent {
-            sender: env.current_contract_address(),
```

```
        sender: recipient.clone(), // attribute mint to the authenticated
                LP (or use env.invoker())
            owner: recipient.clone(),
            tick_lower,
            tick_upper,
            amount,
            amount0,
            amount1,
        }
        .publish(&env);
```

Sushi: Resolved with PR-209. We also have reflected this sender, receiver pattern in the Periphery Repo as well in PR-117 which also resovles Audit Issue Incorrect liquidity position ownership model breaks NFT transfers and operator approvals #28 . Also, important point to note is that There is no such thing as `env.invoker()` in the current SDK, as the audit issue suggests. The best way is to just pass the Address as the parameter and then `require_auth` it.

Zenith: Verified.

[L-2] Protocol fee dust is permanently locked by “keep slot non-empty” logic

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- lib.rs

Description:

In `collect_protocol()`, the withdrawal amount is intentionally reduced by one unit when it equals the full recorded balance: the code applies `amountX -= 1` whenever `amountX == protocol_fees.tokenX`. This “keep slot non-empty” policy creates a pathological case when the remaining balance is exactly 1 unit. A full-withdraw attempt sets `amountX` to 0, subtracts 0 from storage, and transfers 0 — leaving `protocol_fees.tokenX = 1`. Any subsequent call that again requests the full remaining amount will repeat the same branch and still transfer 0, permanently stranding 1 unit in accounting. There is no alternate path that clears this residual, and access control does not mitigate the logical dead-end. The result is a perpetual dust balance and skewed accounting.

Recommendations:

Remove the “keep slot non-empty” decrement so full withdrawals zero the recorded balance. If you must retain a non-empty-slot optimization, guard it so it never triggers when the remaining balance is 1.

```
diff --git a/contracts/dex-pool/src/lib.rs b/contracts/dex-pool/src/lib.rs
-- a/contracts/dex-pool/src/lib.rs
++ b/contracts/dex-pool/src/lib.rs
@@ -463,12 +463,8 @@ pub fn collect_protocol(
        if amount0 > 0 {
            if amount0 == protocol_fees.token0 {
                amount0 -= 1; // keep slot non-empty for storage-gas savings
            }
            protocol_fees.token0 -= amount0;
        if amount0 > 0 {
```

```
            protocol_fees.token0 = protocol_fees.token0.saturating_sub(
                amount0);
             Self::transfer_token(
                 env,
                 &env.current_contract_address(),
                 &recipient,
                 amount0 as i128,
                 &Self::token0(env),
             );
         }
@@ -479,12 +475,8 @@ pub fn collect_protocol(
         if amount1 > 0 {
             if amount1 == protocol_fees.token1 {
                 amount1 -= 1; // same logic
             }
             protocol_fees.token1 -= amount1;
         if amount1 > 0 {
             protocol_fees.token1 = protocol_fees.token1.saturating_sub(
                 amount1);
             Self::transfer_token(
                 env,
                 &env.current_contract_address(),
                 &recipient,
                 amount1 as i128,
                 &Self::token1(env),
             );
         }
```

Sushi: Resolved with PR-211. The keeping of the dust doesn't make sense on Soroban since the `slot0` is stored in instance storage which is extended using the `extend_instance_ttl` whenever it is called, and no extra gas is needed really to recreate it since it never gets deleted. Which is why the dust optimization makes sense only on Ethereum, since to recreate data, you need to spend more gas, but that isn't the case since the data on Soroban never gets deleted (unless its a temporary entry).

Zenith: Verified.

[L-3] Multi-hop exact input swap allows negative input amount which translates to exact output swap

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar-periphery/contracts/swap-router/src/multi_hop.rs#L65

Description:

The `swap_exact_input` function in the swap router is intended for executing multi-hop swaps where the user specifies an exact amount of input tokens. However, unlike its single-hop counterpart (`swap_exact_input_single`), this function does not validate that the provided `amount_in` is a positive number.

The `swap_exact_input_single` function calls `validate_single_swap`, which correctly reverts if the input amount is zero or negative. The multi-hop function `swap_exact_input`, on the other hand, lacks this check and forwards the value directly to the pool's `swap` function.

```
57: // Execute swap for this hop
58: // First hop pulls from user; subsequent hops are prefunded at the pool
59: let swap_result = if i == 0 {
60:     // Pool pulls from user and sends to next pool/final recipient
61:     pool_client.swap(
62:         &params.sender,
63:         &recipient,
64:         &zero_for_one,
65:         &amount_out,
66:         &sqrt_price_limit,
67:     )
68: } else {
```

The underlying pool contract interprets a negative swap amount as a request for an **exact output** swap. This means a user can call `swap_exact_input` with a negative `amount_in` and trigger an exact-output swap, which contradicts the function's explicit purpose. This inconsistency can lead to unexpected behavior and might cause tokens to be swapped in a manner not intended by the caller.

Recommendations:

Consider adding a validation check at the beginning of the `swap_exact_input` function to ensure that `params.amount_in` is greater than zero, thereby maintaining consistency with the single-hop swap implementation and preventing unexpected swap behavior.

Sushi: Resolved with PR-120.

Zenith: Verified.

[L-4] `quote_exact_input_single` does not resolve the price limit

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- hyplabs-sushi-stellar-periphery/contracts/swap-router/src/quote.rs#L28

Description:

The `quote_exact_input_single` function in the swap router is intended to provide a quote for a single-hop swap. However, it handles the `sqrt_price_limit` parameter inconsistently compared to the router's actual swap execution logic.

For example, when a user provides a `sqrt_price_limit` of `0` to the `swap_exact_input_single` function, the router interprets this as "no limit" and substitutes it with a direction-specific unbounded limit (`MIN_SQRT_RATIO + 1` for zero-for-one swaps or `MAX_SQRT_RATIO - 1` for one-for-zero swaps). This logic is handled by the `resolve_sqrt_limit` function.

Conversely, the `quote_exact_input_single` function for single-hop quotes would pass a `sqrt_price_limit` of 0 directly to the pool's `quote_exact_input` function without resolving it. This creates a divergence between the quoted outcome and the actual execution result.

```
15: /// Quote exact input single hop
16: pub fn quote_exact_input_single(
17:     env: &Env,
18:     token_in: &Address,
19:     token_out: &Address,
20:     fee: u32,
21:     amount_in: i128,
22:     sqrt_price_limit: U256,
23: ) -> Result<QuoteResult, SwapRouterError> {
24:     let pool = get_pool_address(env, token_in, token_out, fee)?;
25:     let pool_client = Pool**Sushi:**:new(env, &pool);
26:
27:     let zero_for_one = token_in < token_out;
28:     let result = pool_client.quote_exact_input(&zero_for_one, amount_in,
    &sqrt_price_limit);
```

Recommendations:

Consider modifying the `quote_exact_input_single` function to align its behavior with the execution logic by using the existing `sqrt_price_limit` function.

Sushi: Resolved with PR-123.

Zenith: Verified.

[L-5] Potential LP state mismatch allows burning NFT

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- nonfungible_position_manager.rs#L1107-L1109

Description:

The `burn` function in the `NonFungiblePositionManager` contract allows burning the position NFT.

Before burning, it performs a check to ensure the position is empty:

```
1097: // Get position
1098: let position_key = DataKey::Position(token_id);
1099: extend_persistent_ttl(&env, &position_key);
1100: let position: Position = env
1101:     .storage()
1102:     .persistent()
1103:     .get(&position_key)
1104:     .expect("Invalid token ID");
1105:
1106: // Check position is empty
1107: if position.liquidity ≠ 0 || position.tokens_owed0 ≠ 0 ||
    position.tokens_owed1 ≠ 0 {
1108:     panic!("Position not cleared");
1109: }
```

This check exclusively relies on the NFT position manager's state. However, this state can become stale and may not accurately reflect the full value of the position.

The manager's internal representation of a position's `tokens_owed` is only updated when functions like `increase_liquidity`, `decrease_liquidity`, or `collect` are called, which explicitly synchronize the state with the pool. Since LP positions can be managed both through the NFT manager and directly with the pool, a state mismatch is more likely.

Consequently, the check in the `burn` function can pass, allowing the position NFT to be burned, even though there are still uncollected funds, requiring direct interaction with the

pool to retrieve them.

Recommendations:

Consider modifying the `burn` function to query the pool for the current state before allowing a position to be burned.

Sushi: Resolved with PR-130.

Zenith: Verified. Resolved by removing `extend_persistent_ttl(&env, &position_key)`.

[L-6] Fee growth calculation in `NonfungiblePositionManager` may panic due to unchecked subtraction

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- nonfungible_position_manager.rs#L1433-L1438

Description:

The `get_fee_growth_inside` function in the `NonfungiblePositionManager` contract calculates the fee growth within a specified tick range. It does so by subtracting the fee growth outside the lower and upper ticks from the global fee growth.

The subtractions are performed using `sub`, which will panic on underflow.

```
1433:  fee_growth_global0_x128
1434:      .sub(&lower_fee_growth_outside0)
1435:      .sub(&upper_fee_growth_outside0),
1436:  fee_growth_global1_x128
1437:      .sub(&lower_fee_growth_outside1)
1438:      .sub(&upper_fee_growth_outside1),
```

An underflow can occur if `lower_fee_growth_outside` or `upper_fee_growth_outside` is greater than the global fee growth value. This can happen if the global fee growth (a fixed-point value) wrapped around after reaching the maximum value (`U256`).

If this calculation panics, it would cause functions that rely on it, such as `position_fees`, to fail, preventing users from being able to check their claimable fees.

Recommendations:

Consider using fixed-point wrapping arithmetic for the fee growth calculations to handle potential overflows gracefully.

Sushi: Resolved with PR-111. We have used `wrapping_sub` from the `FixedPoint128` library of the Core repo for the wrapping arithmetic.

Zenith:

[L-7] Pool's reverse lookup key may expire

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- hyplabs-sushi-stellar/contracts/dex-factory/src/pool.rs#L158

Description:

The `create_pool` function in the factory contract stores the address of a new pool under two distinct keys: `(token_a, token_b, fee)` and `(token_b, token_a, fee)`. This design correctly ensures that the pool can be discovered regardless of the order in which the token addresses are provided.

However, the `get_pool` function only extends the Time-To-Live (TTL) for the specific key used in the lookup. If a pool is consistently accessed using only one of the two possible token orderings, the TTL for the reverse-order key will not be refreshed.

Consequently, this reverse key could expire and be removed from storage, breaking the invariant that the pool is accessible with either token ordering without re-creation.

Recommendations:

Consider extending the TTL for both the direct and reverse pool lookup keys whenever a pool is accessed via `get_pool`. This will ensure that both keys remain valid as long as the pool is in use, preserving the intended lookup symmetry.

Sushi: Resolved with PR-199.

Zenith: Verified.

[L-8] Oracle price data has lower granularity than Sushi Swap V3 Solidity implementation

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/oracle.rs#L152-L158

Description:

The oracle in the Solidity implementation of Sushi Swap V3 Solidity implementation writes new price data at most once per block. This ensures that the time-weighted average price (TWAP) reflects changes from every block that contains a swap.

The Soroban implementation, however, throttles oracle writes by enforcing a `MIN_OBSERVATION_INTERVAL` of 12 ledgers (approximately 1 minute) between updates. The `write` function will skip updates if the minimum interval has not passed since the last observation.

If multiple swaps occur within this 1-minute interval, only the first one will trigger an oracle update. Subsequent swaps and their price impact within that window will not be recorded. This results in a coarser time series of price data compared to the Solidity implementation.

Consequently, the oracle's reduced fidelity can lead to less accurate TWAPs due to increased interpolation between fewer data points, especially during periods of high market volatility.

Recommendations:

Consider reducing the `MIN_OBSERVATION_INTERVAL`, aligning the behavior more closely with that of Sushi Swap v3 in Solidity.

Sushi: Resolved with PR-200.

Zenith: Verified.

4.4 Informational

A total of 11 informational findings were identified.

[I-1] Use overflow-safe oracle binary search calculation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/oracle.rs#L530

Description:

The Oracle binary search implementation for historical checkpoints within the `find_surrounding_observations` function calculates the midpoint using the expression (left + right) / 2. Both `left` and `right` are `u32` integers. If their sum were to exceed `u32::MAX`, this operation would overflow and cause a panic, effectively creating a denial-of-service condition for Oracle queries that depend on historical data.

While triggering this overflow is practically infeasible—it would require the number of checkpoints to grow to over two billion, which would take more than 200,000 years at the current rate. However, the binary search for recent observations in the same function correctly uses the overflow-resistant pattern `left + (right - left) / 2`, which is the recommended best practice for such calculations.

Recommendations:

Consider replacing the midpoint calculation in line 530 with the safer alternative.

```
let mid = left + (right - left) / 2;
```

This change eliminates the theoretical overflow vulnerability, improves code consistency, and aligns with secure development best practices.

Sushi: Resolved with PR-208.

Zenith: Verified.

[I-2] Outdated comment in Oracle's `write` function refers to previous tick instead of current tick

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/dex-pool/src/oracle.rs#L169

Description:

A comment in line `169` incorrectly describes the logic for calculating the cumulative tick value.

```
169: // Use the previous tick (stored in last observation) to calculate
    accumulation
170: let current_tick_i64 = tick as i64;
```

However, the implementation calculates `new_tick_cumulative` by using the `tick` parameter passed to the `write` function, which represents the **current** tick. The tick from the `last_observation` is not used for this calculation.

This discrepancy between the comment and the implementation can be misleading and is likely a leftover from a previous implementation where `last_observation.tick` was used.

Recommendations:

Consider updating the comment to accurately reflect that the current tick is used for the cumulative calculation.

Sushi: Resolved with PR-212.

Zenith: Verified.

[I-3] Superfluous `observe_single` call in swap functions

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/dex-pool/src/swap.rs#L394-L399
- hyplabs-sushi-stellar/dex-pool/src/swap.rs#L505-L510

Description:

The `swap` and `swap_prefunded` functions each contain a call to `oracle::Oracle::observe_single`. This call is made when a swap crosses a tick, but it has no side effects and serves no purpose.

```
394: // Perform deferred oracle observation if needed
395: if core.deferred_oracle_tick.is_some() {
396:     // Align tick and liquidity with start-of-swap values, regardless
    of zero liquidity
397:     let start_tick = core.pool_state.slot0.tick;
398:     oracle::Oracle::observe_single(&env, 0, start_tick,
    core.pool_state.liquidity).ok();
399: }
```

The function `oracle::Oracle::observe_single` is a read-only function that computes and returns oracle data for a specific point in time, which are immediately discarded with `.ok()`.

This leads to redundant storage reads and computations, wasting execution resources without affecting the pool's state. The state-mutating function `oracle::Oracle::write` is correctly called earlier in both functions to record the new observation when a tick is crossed. The subsequent call to `observe_single` is therefore superfluous.

Recommendations:

Consider removing the block of code calling `oracle::Oracle::observe_single` in both the `swap` and `swap_prefunded` functions.

Sushi: Resolved with PR-214.

Zenith: Verified.

[I-4] Redundant use of `authorize_as_current_contract` for sub-contract calls

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/dex-factory/src/pool.rs#L108-L118
- hyplabs-sushi-stellar/dex-factory/src/router.rs#L55-L65

Description:

The `set_pool_router_authorized` function unnecessarily uses `authorize_as_current_contract` to authorize a call to the pool contract's `set_router_authorized` function.

```
108:  e.authorize_as_current_contract(vec![
109:      &e,
110:      InvokerContractAuthEntry::Contract(SubContractInvocation {
111:          context: ContractContext {
112:              contract: pool_address.clone(),
113:              fn_name: Symbol::new(&e, "set_router_authorized"),
114:              args: (factory_addr.clone(), default_router.clone(),
    true).into_val(&e),
115:          },
116:          sub_invocations: vec![&e],
117:      }),
118:  ]);
```

Similarly, in `dex-factory/src/pool.rs`, the `create_pool` function also uses `authorize_as_current_contract` for a sub-contract call to `set_router_authorized`.

In Soroban, when a contract calls another contract directly, the authorization is implicitly carried forward from the original invoker. The `authorize_as_current_contract` function is only needed to authorize deeper calls that originate from the next contract call from the current contract.

Since these are direct sub-contract calls, the authorization wrapper is redundant and adds unnecessary code complexity.

Recommendations:

Consider removing the `authorize_as_current_contract` calls in both `set_pool_router_authorized` and `create_pool` as they are not required for direct sub-contract invocations.

Sushi: Resolved with PR-216.

Zenith: Verified.

[I-5] Redundant pool unlock on error paths due to transaction atomicity

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/mint.rs#L94-L96

Description:

The pool contract implements a reentrancy guard by setting `pool_state.slot0.unlocked = false` at the beginning of sensitive operations like `mint`, `burn`, `flash`, and `swap`.

However, on multiple error paths within these functions, the code explicitly sets `pool_state.slot0.unlocked = true` just before returning an `Err`.

This manual unlock operation is redundant. Soroban's execution model guarantees atomic transactions. When a contract entry point returns an `Err`, the framework automatically reverts all state changes made during that transaction. This includes the initial write that locked the pool (`unlocked = false`). Therefore, the subsequent write to unlock it is unnecessary as it will also be reverted, and the pool's state will correctly return to being unlocked.

This pattern is present multiple locations. While this does not introduce a vulnerability, it adds unnecessary code complexity and storage writes (which are ultimately rolled back).

Recommendations:

Consider removing the explicit pool unlocking logic from all error-handling branches that conclude by returning an `Err`. The lock should only be cleared on the successful execution path.

Sushi: Resolved with PR-217.

Zenith: Verified.

[I-6] Operator approval is not cleared on regular NFT transfer

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- nonfungible_position_manager.rs#L986-L995

Description:

The `transfer()` function in the `NonFungiblePositionManager` contract is responsible for transferring ownership of an NFT token. However, contrary to the `transfer_from` function, it does not clear any existing operator approvals for the token being transferred.

When an NFT is transferred, existing approvals for operators, which are authorized addresses that can manage the token on behalf of the owner, should be revoked. The current implementation only updates token ownership but does not reset the approved operator.

This can lead to a situation where a previously approved operator for the old owner retains control over the token even after it has been transferred to a new owner. The new owner may be unaware of this existing approval, posing a security risk as the old operator could still perform actions on their behalf, such as removing liquidity.

Recommendations:

Consider modifying the `transfer` function to clear any existing operator approvals for the token being transferred.

Sushi: Resolved with PR-126. This is a non-issue since there is no operator field to clear, since we entirely rely on the Base NFT approvals, and inside the Base NFT when we do call the transfer() and transfer_from(), any approvals are cleared anyways

Zenith: Verified.

[I-7] Oracle initialization does not store a recent observation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/oracle.rs#L99-L137

Description:

The `initialize` function in `contracts/dex-pool/src/oracle.rs` sets up the oracle by creating an initial observation. This observation is stored as the latest observation and as the first historical checkpoint at index 0. However, it is not stored as a "recent observation".

The first recent observation is stored later when `write` is called, starting at index 1. Because no recent observation is stored at index 0 during initialization, any binary search performed on recent observations may waste reads by probing index 0, which will always be empty. While this does not affect the correctness of the oracle, it introduces a minor inefficiency.

Recommendations:

Consider adding a recent observation entry during the `initialize` function, similar to how the historical observation is stored.

Sushi: Resolved with PR-196.

Zenith: Verified.

[I-8] Inconsistent operator approval mechanism and lack of `approve_for_all` support in the `NonfungiblePositionManager` contract

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- nonfungible_position_manager.rs

Description:

The `approve` function in the `NonfungiblePositionManager` contract updates the approval status for a single NFT by calling `Base::approve` and simultaneously sets an `operator` field on the `Position` struct. This creates two sources of truth for approvals:

1. The `Base` NFT contract's approval state and
2. The `operator` field within each position

While not explicitly harmful, this redundancy can lead to inconsistencies and increase contract complexity.

Furthermore, the contract only supports approvals for individual token IDs and lacks an `approve_for_all` function. This deviates from standard NFT implementations and harms the user experience, as it prevents users from approving a single operator for all their positions in a single transaction.

Recommendations:

Consider removing the `operator` field from the `Position` struct and relying entirely on the `Base` NFT contract's approval state as the single source of truth. Additionally, consider adding `approve_for_all` and `is_approved_for_all` functions to align with standard NFT behavior and improve the contract's flexibility and usability.

Sushi: Resolved with PR-126.

Zenith: Verified.

[I-9] Hardcoded value for observation TTL reduces readability

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/oracle.rs#L466-L468

Description:

In `contracts/dex-pool/src/oracle.rs#L466-L468`, a hardcoded value of `17280` is used to determine the oldest valid index for a recent observation.

```
466:  let oldest_valid_index = metadata
467:      .recent_observation_index
468:      .saturating_sub(17280 - ttl_buffer);
```

The use of this "magic number" without explanation obscures its purpose, making the code more difficult to understand and maintain.

Recommendations:

It is recommended to replace the magic number with the `RECENT_OBSERVATION_TTL` constant.

Sushi: Resolved with PR-197.

Zenith: Verified.

[I-10] Redundant TTL extensions in tick management

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/tick.rs#L152
- hyplabs-sushi-stellar/contracts/dex-pool/src/tick.rs#L193

Description:

The `tick::update` function, which manages the lifecycle of tick data in persistent storage, contains two calls to `extend_persistent_ttl`. One call is made after reading tick data, and the other is made after writing it.

The purpose of `extend_persistent_ttl` is to prolong the storage entry's lifetime. It is implemented as an optimization that only extends the TTL if it is within one month of expiring. However, the `tick::update` function subsequently writes to the same storage entry in line 190 using `env.storage().persistent().set(...)`.

In Soroban, a `set` operation on a persistent storage entry unconditionally resets its Time-to-Live (TTL) to the maximum configured value. Consequently, both calls to `extend_persistent_ttl` within this function are redundant:

1. The first call in line 152 is unnecessary because the `set` operation in line 190 will refresh the TTL anyway.
2. The second call in line 193 occurs immediately after the `set` operation has already reset the TTL to its maximum. The conditional check within `extend_persistent_ttl` will therefore not pass, and the underlying `extend_ttl` operation will not be performed.

While the logic is sound, these redundant calls incur unnecessary execution costs without contributing to the contract's state longevity.

Similarly, this pattern is observed in other parts of the codebase where storage entries are updated and TTLs are explicitly extended, such as the `tick::cross` function.

Recommendations:

Consider removing both calls to `extend_persistent_ttl` in the `tick::update` function. The single `env.storage().persistent().set` call is sufficient to ensure the tick entry's TTL is

reset upon update, which simplifies the code and reduces transaction costs.

Sushi: Resolved with PR-133 - which removes the redundant extensions from the `add_token_to_user` and `remove_token_from_user` in this PR for the Periphery, PR-237 - to call `extend_persistent_ttl` after storing `TickBitMapKey::TickBitmap` in the persistent storage, and `@81e33f8f59 ...` - for some small optimization in the `flip_tick code`.

Zenith: Verified.

[I-11] Redundant explicit locking mechanisms

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- hyplabs-sushi-stellar/contracts/dex-pool/src/swap.rs#L169

Description:

The DEX pool utilizes two distinct locking mechanisms to prevent reentrancy: a boolean `unlocked` flag within the `Slot0` storage struct, and a presence-based lock using a `LOCK_KEY` in instance storage.

However, these explicit locking mechanisms are redundant and not necessary on the Soroban platform. Unlike EVM-based chains, where reentrancy is a primary concern due to the execution model, Soroban's host environment enforces strict rules regarding contract invocation and natively prevents cross-contract reentrancy calls. The host environment prevents a contract from being re-entered while it is already executing in the current call stack.

If an external call (e.g., to a token contract) attempts to call back into the DEX pool, the transaction will fail at the host level before reaching the contract logic.

The existence of these locks introduces unneeded complexity and overhead (due to additional storage writes) without providing additional security.

Recommendations:

Consider removing all explicit reentrancy locking mechanisms.

Sushi: Resolved with PR-241 by removing the explicit locking mechanisms while keeping the flash locking since it is still needed.

Zenith: Verified. For flash loan, the locking mechanism is still needed because it consists of two separate calls, which must be called sequentially and by the same initiator.