Security Review Report
NM-0565 Skate AMM

NETHERMIND
SECURITY

(September 16, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for Skate. Skate AMM is a cross-chain automated market maker designed to establish a unified liquidity state across multiple blockchain ecosystems, leveraging the mathematical principles of Uniswap V3's concentrated liquidity model.

The protocol features a novel hybrid architecture that separates core logic from user interactions. A central Kernel Chain (the Skate chain) serves as the single source of truth for all AMM calculations and accounting, while multiple Periphery Chains (including EVM chains and Solana) host user-facing `PeripheryPool` contracts that hold assets in custody.

User-initiated operations, such as swaps or liquidity provisions, are staged on a `PeripheryPool` and then relayed by off-chain actors called Executors to the Kernel for processing. The Kernel computes the outcome and issues a "Task," which the Executor then executes back on the Periphery Chain to finalize the transaction and settle user funds. This hybrid model places significant trust in the off-chain Executor system, making the security and reliability of this centralized infrastructure as critical as the on-chain smart contracts themselves for the protocol's overall safety.

The scope of this security review encompassed the on-chain implementation of the Skate AMM, including the Kernel contracts on the Skate chain, the Periphery contracts for EVM-compatible chains, and the Periphery program for Solana.

The audit comprises 4090 lines of Solidity code and 1739 lines of Rust code. The audit was performed using (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 33 points of attention, where three are classified as `Critical`, seven are classified as `High`, one is classified as `medium`, two are classified as `Low` and twenty are classified as `Informational` or `Best Practices` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test output. Section 9 concludes the document.

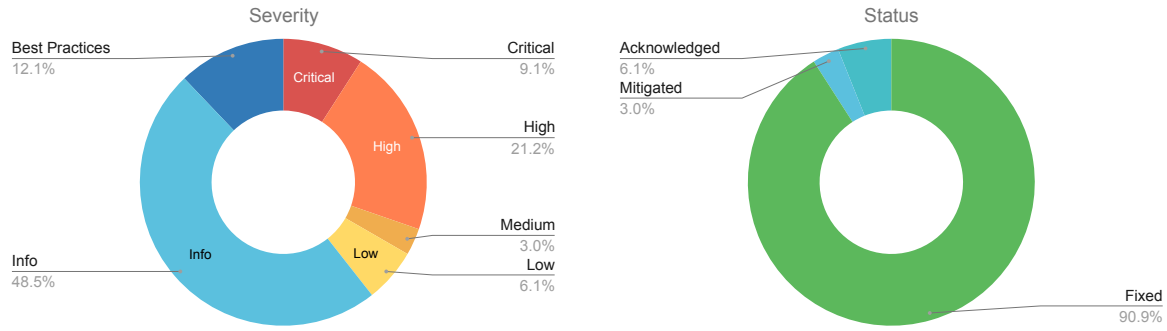

Fig. 1: Distribution of issues: Critical (3), **High** (7), **Medium** (1), **Low** (2), **Undetermined** (0), **Informational** (16), **Best Practices** (4). **Distribution of status: Fixed** (30), **Acknowledged** (2), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	July 29, 2025
Final Report	September 16, 2025
Repository	Skate-Org/Skate-AMM
Initial Commit	42bf0980e5cd77f4063ac1038a8abd7dd08869ab
Final Commit	0d0131e7fe67147a48f72e5f750bb64a224003f4
Documentation	Notion page with Skate AMM overview
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/skate/kernel/MessageBox.sol	123	5	4.1%	20	148
2	src/skate/kernel/AccountStorage.sol	55	10	18.2%	6	71
3	src/skate/kernel/SkateApp.sol	75	2	2.7%	13	90
4	src/skate/kernel/AccountRegistry.sol	148	2	1.4%	17	167
5	src/skate/kernel/interfaces/IAccountRegistry.sol	62	69	111.3%	11	142
6	src/skate/kernel/interfaces/IMessageBox.sol	56	39	69.6%	15	110
7	src/skate/kernel/interfaces/ISkateApp.sol	18	30	166.7%	9	57
8	src/skate/common/IExecutorRegistry.sol	12	25	208.3%	6	43
9	src/skate/common/ExecutorRegistry.sol	35	1	2.9%	10	46
10	src/skate/common/Multicall.sol	48	1	2.1%	9	58
11	src/skate/common/utils/Utils.sol	16	6	37.5%	7	29
12	src/skate/periphery/ActionBox.sol	47	51	108.5%	12	110
13	src/skate/periphery/SkateGateway.sol	88	3	3.4%	20	111
14	src/skate/periphery/SkateAppPeriphery.sol	22	1	4.5%	7	30
15	src/skate/periphery/interfaces/ISkateAppPeriphery.sol	5	7	140.0%	2	14
16	src/skate/periphery/interfaces/ISkateGateway.sol	36	47	130.6%	11	94
17	src/amm/kernel/KernelPool.sol	661	80	12.1%	102	843
18	src/amm/kernel/KernelEventEmitter.sol	162	1	0.6%	17	180
19	src/amm/kernel/KernelManagerStorage.sol	88	1	1.1%	13	102
20	src/amm/kernel/KernelManager.sol	254	2	0.8%	33	289
21	src/amm/kernel/libraries/KernelManagerLib.sol	833	42	5.0%	77	952
22	src/amm/kernel/libraries/DataTypes.sol	45	10	22.2%	7	62
23	src/amm/kernel/interfaces/IKernelManager.sol	110	1	0.9%	28	139
24	src/amm/kernel/interfaces/IERC20Minimal.sol	16	31	193.8%	7	54
25	src/amm/kernel/interfaces/IKernelPool.sol	15	5	33.3%	2	22
26	src/amm/kernel/interfaces/pool/IUniswapV3PoolActions.sol	56	54	96.4%	6	116
27	src/amm/kernel/interfaces/pool/IUniswapV3PoolImmutables.sol	8	18	225.0%	5	31
28	src/amm/kernel/interfaces/pool/IUniswapV3PoolState.sol	56	55	98.2%	12	123
29	src/amm/kernel/interfaces/pool/IUniswapV3PoolDerivedState.sol	21	23	109.5%	2	46
30	src/amm/kernel/interfaces/pool/IUniswapV3PoolEvents.sol	68	1	1.5%	9	78
31	src/amm/kernel/interfaces/pool/IUniswapV3PoolOwnerActions.sol	11	12	109.1%	2	25
32	src/amm/periphery/PeripheryManager.sol	118	7	5.9%	16	141
33	src/amm/periphery/PeripheryPool.sol	458	10	2.2%	56	524
34	src/amm/periphery/PeripheryEventEmitter.sol	164	14	8.5%	24	202
35	src/amm/periphery/interfaces/IPeripheryPool.sol	65	1	1.5%	21	87
36	src/amm/periphery/interfaces/IPeripheryManager.sol	35	1	2.9%	15	51
37	solana/src/lib.rs	1595	243	15.2%	144	1982
38	solana/src/kernel_action.rs	94	3	3.2%	3	100
39	solana/src/helper.rs	50	20	40.0%	13	83
	Total	**5829**	**934**	**16.0%**	**789**	**7552**

3 Summary of Issues

	Finding	Severity	Update
1	Insecure validation in `PeripheryEventEmitter` allows off-chain backend to be deceived	Critical	Fixed
2	Missing authorization check in `update_pool_config(...)` allows full drain of funds from periphery pools	Critical	Fixed
3	Race condition in `removeStagedAssets` allows for theft of any staged assets	Critical	Fixed
4	Inconsistent `extraData` normalization leads to potential loss of funds	High	Fixed
5	Incorrect decimal normalization in swap leads to bypassed slippage protection	High	Fixed
6	Incorrect liquidity recalculation leads to loss of funds	High	Fixed
7	Ineffective Slippage Protection and Denial of Service for Exact-Output Swaps	High	Fixed
8	Rounding in favor of the user in `normalize_amount_for_transfer` leads to periphery pool insolvency	High	Fixed
9	Swap logic incorrectly handles exact output swaps leading to failed slippage protection	High	Fixed
10	The `initialize_tokens` instruction lacks access control and validation	High	Fixed
11	Unsafe usage of ERC-20 token transfer functions	Medium	Fixed
12	Dust amounts will get stuck in `PeripheryPool` contracts	Low	Fixed
13	Exact output swaps may revert unexpectedly when a price limit is reached	Low	Fixed
14	Fee Collection Mechanism Lacks Granularity	Info	Acknowledged
15	Incorrect accounting of global staged assets in `swap(...)`	Info	Fixed
16	Missing Uniswap V3 standard parameters like `deadline` and `recipient`	Info	Fixed
17	Only NFT position owner can increase liquidity	Info	Fixed
18	Potential `actionId` collision in `collectProtocol(...)`	Info	Mitigated
19	Refunded tokens from swaps are sent to the recipient address	Info	Fixed
20	Swap behavior deviates from Uniswap V3 regarding `sqrtPriceLimitX96`	Info	Fixed
21	The `_normalizeTokenAmount(...)` function does not handle decimals larger than `18`	Info	Fixed
22	The `burn(...)` function lacks a check for available liquidity	Info	Fixed
23	The `changePeripheryPool(...)` function can add new periphery pools	Info	Fixed
24	The `createPool` function does not prevent creating a pool with identical tokens	Info	Fixed
25	The `settle_swap(...)` function does not utilize the `recipient` for settlement	Info	Fixed
26	The `swap(...)` function does not validate the destination VM type	Info	Fixed
27	Tokens without a `decimals()` function are not supported	Info	Fixed
28	Unsafe cast to `int256` in the `swap(...)` function	Info	Fixed
29	`generate_action_id(...)` may return the same hash for different actions	Info	Fixed
30	Missing `extraData` validation in `mint(...)` and `increaseLiquidity(...)`	Best Practices	Fixed
31	The `createPool(...)` function does not validate if `tickSpacing` is configured	Best Practices	Fixed
32	The `setFeeAmountTickSpacing(...)` function lacks input validation	Best Practices	Fixed
33	Usage of `tx.origin` for authorization is unsafe	Best Practices	Acknowledged

4 System Overview

The Skate AMM protocol is a cross-chain automated market maker designed to create a unified liquidity state across multiple blockchain ecosystems. It leverages the mathematical principles of Uniswap V3's concentrated liquidity model while introducing a novel architecture that separates user interactions from core logic. The system is composed of a central `Kernel Chain` (the Skate chain), which serves as the single source of truth for AMM logic and accounting, and multiple `Periphery Chains` (such as various EVM chains and Solana), where users deposit assets and initiate transactions.

This architecture is facilitated by key smart contracts on each layer. The Kernel Chain hosts the `KernelManager`, which orchestrates operations and manages liquidity positions as NFTs, and the `KernelPool` contracts, which contain the core AMM logic. On each Periphery Chain (EVM only), a `PeripheryManager` acts as a factory for deploying `PeripheryPool` contracts. These user-facing `PeripheryPool` contracts hold all user-deposited assets in custody. Communication between the periphery and the kernel is managed by off-chain actors known as Executors.

4.1 User Actions and Asset Staging on Periphery Chains

All user-initiated AMM operations, such as providing liquidity (`mint`, `increase liquidity`), removing liquidity (`burn`, `decrease liquidity`), or trading tokens (`swap`), begin on a `PeripheryPool` contract/program on the user's chain of choice. For EVM, the `PeripheryManager` contract is responsible for deploying these pools for specific token pairs and fee tiers, whereas for Solana, `initialize_pool(...)` and `initialize_tokens(...)` in `PeripheryPool` program is responsible for creating `PeripheryPool` accounts for specific token pairs and fee tiers.

A key concept in this stage is "asset staging". When a user performs an action that requires them to provide tokens (e.g., a deposit for `mint` or the input for a `swap`), the assets are transferred from the user's wallet directly into the custody of the `PeripheryPool` contract. The contract then records the user's contribution in a local, user-specific storage mapping (e.g., `usersData` on EVM or a `UserDataAccount` PDA on Solana). This staged balance represents a temporary credit that is reconciled upon the completion of the cross-chain operation.

4.2 Cross-Chain Action Propagation

Once a user's action is initiated and their assets are staged, the `PeripheryPool` prepares a message for the Kernel Chain. This process involves several steps to ensure standardized communication:

- **Token Amount Normalization**: The Kernel Chain expects all token amounts to be in a standard 18-decimal format for consistent calculations. The handling of this normalization varies by periphery implementation. On EVM chains, the `PeripheryPool` contract normalizes token amounts on-chain before packaging the action. On Solana, the on-chain program emits action data with unnormalized amounts, delegating the responsibility of normalization to the off-chain backend system, which must convert the values before relaying them to the Kernel.
- **Action Packaging**: The user's intent, including the operation type, parameters, and (on EVM) normalized amounts, is encoded into a standardized data structure. This structure is known as an `Action` on EVM chains or a `KernelAction` on Solana.
- **Submission to ActionBox**: The encoded action is submitted to a local `ActionBox` contract. The `ActionBox` functions as a permissionless, on-chain outbound message queue, creating a verifiable record of the user's intent.
- **Event Emission**: Concurrently, a `PeripheryEventEmitter` contract (EVM only) emits an event detailing the action. This event serves as a signal for the off-chain infrastructure that a new user request is ready to be processed.

4.3 Executor Relay and Kernel Processing

The protocol relies on off-chain actors called **Executors** to monitor and relay messages between chains. An Executor observes the `ActionBox` contract/program and event emissions on all Periphery Chains. Upon detecting a new action, the Executor retrieves the encoded data and is responsible for delivering it to the Kernel Chain.

On the Kernel Chain, the Executor submits the user's action as an "Intent" to the central `MessageBox` contract. This contract authenticates the Executor and forwards the intent to the `KernelManager`. The `KernelManager` decodes the action data and orchestrates the core logic. It interacts with the appropriate `KernelPool` contract, which executes the Uniswap V3-style calculations to determine the outcome of the operation (e.g., calculating the amount of liquidity to mint or the result of a swap).

4.4 Task Generation and Settlement

After processing the action, the `KernelManager` generates one or more **Tasks**. A Task is an executable directive containing the results of the kernel-side computation. For example, a swap task would include the amount of output tokens to be transferred to the user and any unused input tokens to be refunded. These Tasks are logged in the `MessageBox` on the Kernel Chain.

The Executor monitors the `MessageBox` for new Tasks. Upon retrieving a Task, the Executor facilitates its execution on the target Periphery Chain. The settlement process differs slightly based on the chain's implementation:

- On EVM chains, the Executor calls the `executeTask` function on a dedicated `SkateGateway` contract. This contract verifies the Executor and then calls the appropriate privileged settlement function (e.g., `settleMint`) on the originating `PeripheryPool`.
- On Solana, the Executor's address is whitelisted as a `gateway` directly within the `periphery_pool` program's state. The Executor calls the settlement functions (e.g., `settle_mint`) directly on the program.

In both cases, the settlement call "unstages" the user's assets from local accounting and transfers the final assets from the pool's custody to the user. If a user's action fails during kernel processing, an Executor is expected to call the `removeStagedAssets` function on the `PeripheryPool` to refund the user's staged balance. The process is the same for Solana programs.

4.5 Security Assumptions

The protocol's hybrid on-chain and off-chain design relies on several key security assumptions regarding the off-chain infrastructure.

- **Executor Trust**: The security model places significant trust in the off-chain Executor. The correctness of the system's cross-chain accounting and the final settlement of user assets are critically dependent on the Executor's reliable and secure operation.
- **Off-Chain Logic Integrity**: The division of logic between on-chain programs and off-chain services differs by chain. On Solana, the Executor is responsible for correctly normalizing action data before relaying it to the Kernel and, conversely, interpreting values from Kernel Tasks for settlement. This makes the off-chain logic a critical component for maintaining correct accounting.
- **Complex Decimal Handling on Solana**: The Solana implementation introduces a multi-layered decimal conversion process that is managed entirely by the off-chain Executor. User-facing functions on Solana expect input amounts to be pre-normalized to 6 decimals. The on-chain program passes these 6-decimal values to the `ActionBox` while converting them back to the token's native decimals for the actual asset transfer. The Executor is then responsible for converting these 6-decimal action amounts to the Kernel's required 18-decimal standard. For settlement, the Executor must again perform conversions, providing settlement amounts in 6 decimals and transfer amounts in the token's native decimals. Any error in this complex off-chain conversion logic could lead to incorrect accounting and potential loss of funds.
- **Liveness Dependency**: The system does not currently feature a trustless failsafe mechanism for users to withdraw staged assets. If the Executor infrastructure were to experience prolonged downtime, user funds intended for an operation could remain locked in `PeripheryPool` contracts until the service is restored.
- **Communication Reliability**: The end-to-end completion of a user's action relies on the off-chain system successfully picking up both the initial `Action` from the periphery and the resulting `Task` from the Kernel. A failure in this off-chain communication link would cause the user's operation to stall.
- **Periphery Pool Solvency**: The successful settlement of an operation depends on the `PeripheryPool` having sufficient token balances to fulfill the Task issued by the Kernel. If the Kernel executes a task (e.g., a withdrawal) but the corresponding `PeripheryPool` has a liquidity shortfall, the settlement transaction on the periphery will fail and remain pending. The user would be unable to receive their funds until additional liquidity is deposited into that specific pool by other protocol users.

Consequently, the security of the Skate AMM protocol is a function of both the on-chain smart contracts and the integrity and correctness of the centralized backend and Executor logic.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security 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 Security 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.

6 Issues

6.1 [Critical] Insecure validation in `PeripheryEventEmitter` allows off-chain backend to be deceived

File(s): src/amm/periphery/PeripheryEventEmitter.sol

Description: The Skate AMM architecture relies on a cross-chain communication model where actions on periphery chains are relayed to the Skate chain for execution. The `PeripheryPool` contract initiates these actions by creating an entry in the permissionless `ActionBox` and then triggers a corresponding event via the `PeripheryEventEmitter` contract. The security model assumes that the `PeripheryEventEmitter` is a permissioned contract, meaning only legitimate `PeripheryPool` contracts can call it to emit events. These events are used by off-chain executors to validate and process the actions created in the `ActionBox`.

However, the validation mechanism within the `PeripheryEventEmitter` contract is flawed. The internal `_isValidCaller(...)` function checks if the `peripheryPool` address passed as an argument is a valid pool, instead of verifying that `msg.sender` is the one that is a valid pool.

```
function mint(
    address peripheryPool,
    address kernelPool,
    address user,
    int24 tickLower,
    int24 tickUpper,
    uint256 amount0,
    uint256 amount1
) external {
    // @audit The check validates the argument, not the actual caller (msg.sender).
    _isValidCaller(peripheryPool);
    emit Minted(peripheryPool, kernelPool, user, tickLower, tickUpper, amount0, amount1);
}
```

The `_isValidCaller(...)` function checks against the `poolExists` mapping in the `PeripheryManager` contract.

```
function _isValidCaller(address peripheryPool) private view {
    // @audit-issue The function confirms that `peripheryPool` is a valid pool,
    // but does not check if `msg.sender` is this pool.
    require(manager.poolExists(peripheryPool));
}
```

This vulnerability allows an attacker to call any function in the `PeripheryEventEmitter` from a malicious contract. By providing the address of a legitimate `PeripheryPool` as the `peripheryPool` argument, the attacker can successfully pass the `_isValidCaller(...)` check. This enables the attacker to emit fraudulent events (e.g., `Minted`, `Swapped`) with arbitrary parameters, such as minting a large number of tokens to their address without providing any underlying assets. These spoofed events appear to originate from a legitimate pool, breaking the backend's core security assumption that only valid pools can emit events from this contract. The off-chain executor, which trusts events from the `PeripheryEventEmitter`, will be deceived by these fraudulent events, leading to the execution of unauthorized actions on the Skate chain and resulting in a direct loss of funds from the AMM pools.

Recommendation(s): Consider redesigning the validation mechanism within the `PeripheryEventEmitter` contract to authenticate the caller (`msg.sender`) instead of an input parameter.

Status: Fixed

Update from the client: The remediation is applied. commit

6.2 [Critical] Missing authorization check in `update_pool_config(...)` allows full drain of funds from periphery pools

File(s): solana/src/lib.rs

Description: The `update_pool_config(...)` function is intended to be a privileged function that allows the pool's manager to update critical configuration parameters. These parameters include the `gateways` array, which contains a list of public keys authorized to call sensitive settlement functions.

However, the account validation for this function is insufficient. While the `UpdatePoolConfig` struct correctly specifies that the `manager` account must be a `Signer`, it fails to verify that this signer is the _actual manager_ of the `periphery_pool`. The check to ensure that `ctx.accounts.manager.key()` is the same as `ctx.accounts.periphery_pool.manager` is missing.

```
pub fn update_pool_config(
    ctx: Context<UpdatePoolConfig>,
    chain_id: Option<u32>,
    src_vm_type: Option<u8>,
    description: Option<String>,
    gateways: Option<Vec<Pubkey>>,
    kernel_factory: Option<String>,
    kernel_pool: Option<String>,
) -> Result<()> {
    // @audit-issue The function lacks a check to ensure that the signer
    // is the legitimate manager of the pool.
    let pool = &mut ctx.accounts.periphery_pool;
    // ...
    if let Some(gateways) = gateways {
        pool.gateways = gateways;
    }
    // ... other updates
    Ok(())
}
```

The corresponding `UpdatePoolConfig` struct is defined as follows:

```
#[derive(Accounts)]
pub struct UpdatePoolConfig<'info> {
    #[account(mut)]
    pub periphery_pool: Box<Account<'info, PeripheryPool>>,

    // @audit-issue The `has_one = manager` constraint is missing on the
    // `periphery_pool` account.
    #[account(mut)]
    pub manager: Signer<'info>,
}
```

This vulnerability allows any malicious actor to call `update_pool_config(...)`. An attacker can add their own public key to the `gateways` list. As an authorized gateway, the attacker can then invoke privileged functions such as `settle_mint(...)`, `settle_burn(...)`, and `settle_swap(...)`. These functions execute token transfers from the pool's token accounts. By providing crafted data to these settlement functions, the attacker can drain all `token0` and `token1` funds held by the `PeripheryPool` contract.

Recommendation(s): Consider enforcing that the `manager` signer account matches the manager address stored in the `periphery_pool` account. This can be achieved by adding the `has_one = manager` constraint to the `periphery_pool` account in the `UpdatePoolConfig` validation struct.

Status: Fixed

Update from the client: Fixed in solana commit.

6.3 [Critical] Race condition in `removeStagedAssets` allows for theft of any staged assets

File(s): src/amm/periphery/PeripheryPool.sol

Description: The Skate AMM architecture separates user interactions on periphery chains from core logic on the Skate chain. When a user calls functions like `mint(...)`, `increaseLiquidity(...)`, or `swap(...)` on a `PeripheryPool` contract, their tokens are transferred to the pool, and their staged balance is recorded in the `usersData` mapping. An off-chain executor then picks up this action and attempts to process it on the `KernelManager` contract on the Skate chain.

If the kernel transaction succeeds, an appropriate settlement function is called on the `PeripheryPool` to finalize the state. However, if the kernel transaction fails for any reason, an executor is expected to call `removeStagedAssets(...)` to refund the user their staged tokens.

The vulnerability lies in the `removeStagedAssets(...)` function. This function refunds the user's _entire_ staged balance, as recorded in `usersData`, rather than the specific amounts associated with the single failed action. This creates a critical race condition that a malicious user can exploit across any function that stages assets.

An attacker can initiate a transaction (e.g., a `mint` or `swap`) that is designed to fail on the kernel chain. When they observe an executor submitting a transaction to call `removeStagedAssets(...)` to refund them, they can front-run the executor's transaction with a second, valid transaction (e.g., another `mint`, `increaseLiquidity`, or `swap`). This second call adds more funds to their staged balance in `usersData`. The subsequent execution of `removeStagedAssets(...)` will then read the now-inflated balance and transfer the assets from _both_ actions back to the attacker.

The impact is that the attacker receives a full refund for their second (valid) action, while that action is still processed successfully on the kernel chain, granting them a free liquidity position or swap. This drains funds from the `PeripheryPool`, as the assets that should have backed the user's action have been incorrectly returned. This critical issue is compounded by the `_settle(...)` function, which silently handles accounting underflows, masking the state inconsistency when the second action is eventually settled.

All functions that add to a user's `usersData` balance are vectors for this attack, including `mint(...)`:

```
function mint(...) external override {
    // ...
    // @audit The user's staged balance is increased, making it vulnerable.
    UserData storage data = usersData[msg.sender];
    data.amount0 += amount0;
    data.amount1 += amount1;
    // ...
}
```

The `increaseLiquidity(...)` function follows the same vulnerable pattern:

```
function increaseLiquidity(...) external {
    // ...
    // @audit This function also adds to the user's staged balance.
    UserData storage data = usersData[msg.sender];
    data.amount0 += amount0;
    data.amount1 += amount1;
    // ...
}
```

Similarly, the `swap(...)` function stages the input token amount, making it a viable entry point for the attack:

```
function swap(...) external override {
    // ...
    // @audit The input token for the swap is staged here.
    if (zeroForOne) {
        tokenIn = token0;
        usersData[msg.sender].amount0 += amountIn;
    } else {
        tokenIn = token1;
        usersData[msg.sender].amount1 += amountIn;
    }
    // ...
}
```

Regardless of which function is used to stage the assets, the generic `removeStagedAssets(...)` function refunds the entire balance, enabling the exploit.

```solidity
function removeStagedAssets(address usr) external override onlyExecutor {
    // @audit-issue Reads the entire staged balance for the user,
    // not just the balance from a specific failed action.
    UserData memory userData = usersData[usr];
    uint256 amount0 = userData.amount0;
    uint256 amount1 = userData.amount1;
    // @audit This zeroes out the user's balance.
    _settle(usr, amount0, amount1);
    // @audit The user is refunded the entire staged amount.
    if (amount0 != 0) IERC20(token0).transfer(usr, amount0);
    if (amount1 != 0) IERC20(token1).transfer(usr, amount1);
    // ...
}
```

The `_settle(...)` function's silent handling of underflows prevents a revert that would otherwise signal the accounting error when the attacker's valid transaction is settled after the refund has already occurred.

```solidity
function _settle(address user, uint256 amount0, uint256 amount1) private {
    // to avoid under flow due to possible precision mismatch
    UserData storage userData = usersData[user];
    // @audit-issue After `removeStagedAssets` has run, `userData.amount0` will be 0.
    // When the settlement function for the second valid action calls this,
    // it should underflow but instead silently evaluates to 0, hiding the exploit.
    userData.amount0 = userData.amount0 > amount0 ? userData.amount0 - amount0 : 0;
    userData.amount1 = userData.amount1 > amount1 ? userData.amount1 - amount1 : 0;

    manager.eventEmitter().amountSettled(address(this), kernelPool, user, amount0, amount1);
}
```

Recommendation(s): Consider modifying the `removeStagedAssets(...)` function to accept the specific `amount0` and `amount1` to be removed as parameters, likely derived from the failed action data. This will ensure that only the funds from a single failed action are refunded, preventing the described race condition.

It is also recommended to review the settlement logic in the `_settle(...)` function. The current approach to prevent underflow-related reverts inadvertently conceals the theft of funds.

Status: Fixed

Update from the client: The remediation is applied. commit

Update from the Nethermind Security: The applied fix is incorrect as it still uses the amount from the user data storage struct to transfer the tokens to the user. The attack described in the issue is still possible.

Update from the client: Additonal remediation is applied. commit

6.4 [High] Inconsistent `extraData` normalization leads to potential loss of funds

File(s): src/amm/periphery/PeripheryPool.sol, src/amm/kernel/libraries/KernelManagerLib.sol

Description: In the Skate AMM, user interactions occur on `PeripheryPool` contracts deployed on various chains, while the core AMM logic is executed on the central `KernelManager` contract. When a user wishes to withdraw liquidity via the `burn(...)` or `decreaseLiquidity(...)` functions in the `PeripheryPool`, they can provide slippage protection parameters (`amount0Min`, `amount1Min`) within the `extraData` field.

The `PeripheryPool` contract first uses these minimum amounts to check against its own token balances. These balances are in their native, un-normalized decimal format.

```
// src/amm/periphery/PeripheryPool.sol
function decreaseLiquidity(
    uint256 tokenId,
    uint128 liquidityAmount,
    bytes memory extraData // must contain amount0Min, amount1Min
)
    external
{
    // @audit extraData is decoded and used to check against un-normalized balances.
    (uint256 amount0Min, uint256 amount1Min) = abi.decode(extraData, (uint256, uint256));
    (uint256 amount0Available, uint256 amount1Available) = balancesAvailable();
    require(
        amount0Min <= amount0Available && amount1Min <= amount1Available,
        "PeripheryPool::decreaseLiquidity: not enough balance in periphery pool"
    );
    // ...
    // @audit The same extraData is passed to the kernel.
    action.kernelAppCalldata = abi.encodeWithSignature(
        "decreaseLiquidity(bytes)",
        abi.encode(
            // ...
            liquidityAmount,
            extraData
        )
    );
    // ...
}
```

The same `extraData` is then forwarded to the `KernelManager` contract. Inside the `KernelManagerLib`, the `decreaseLiquidity(...)` function performs its own slippage check. However, this check compares the `amount_Min` values from `extraData` against the token amounts returned by the `KernelPool`, which are normalized to `18` decimals.

```
// src/amm/kernel/libraries/KernelManagerLib.sol
function decreaseLiquidity(
    DataTypes.State storage state,
    bytes calldata decreaseLiquidityData
)
    external
    returns (IMessageBox.Task[] memory tasks)
{
    (
        // ...
        uint128 liquidityAmount,
        bytes memory extraData
    ) = abi.decode(
        decreaseLiquidityData,
        (bytes32, uint256, uint256, bytes32, address, uint256, uint128, bytes)
    );
    // ...
    (vars.amount0, vars.amount1) = vars.pool.burn(
        address(this), position.tickLower, position.tickUpper, liquidityAmount, false
    );
    // @audit-issue extraData is decoded again and compared against normalized amounts.
    (vars.amount0Min, vars.amount1Min) = abi.decode(extraData, (uint256, uint256));
    require(
        vars.amount0 >= vars.amount0Min && vars.amount1 >= vars.amount1Min,
        "price slippage check"
    );
    // ...
}
```

This creates a fundamental conflict. It is impossible for a user to provide a single `extraData` value that is valid for both checks simultane-

ously for tokens that do not have `18` decimals.

- If the user provides un-normalized `amount_Min` values (correct for the periphery check), the slippage check on the kernel side becomes ineffective. An attacker could exploit this by front-running the transaction, manipulating the price, and causing the user to receive far fewer tokens than expected, leading to significant financial loss;
- If the user provides normalized `amount_Min` values (correct for the kernel check), the initial check on the periphery side will fail for any token with fewer than `18` decimals. This is because the check compares a numerically large, normalized value (e.g., `1 * 10^18` for `1` token) against the pool's total balance in its native, smaller decimal format (e.g., `1000 * 10^6` for `1000` USDC). The check will incorrectly determine that the minimum amount requested exceeds the available balance, causing legitimate transactions to revert and preventing users from withdrawing liquidity with proper slippage protection;

Recommendation(s): Consider revisiting and changing the slippage check mechanism in such a way that appropriate checks, using the correct decimal precision, are performed on each chain. It is also recommended to review all other functions that accept `extraData` to ensure consistent handling of slippage parameters across the entire protocol.

Status: Fixed

Update from the client: The remediation is applied. commit

6.5 [High] Incorrect decimal normalization in swap leads to bypassed slippage protection

File(s): src/amm/periphery/PeripheryPool.sol, src/amm/kernel/libraries/KernelManagerLib.sol

Description: The Skate AMM architecture separates user interactions on periphery chains from the core logic on the Skate chain. The `PeripheryPool` contract, deployed on various EVM chains, is responsible for normalizing all token amounts to a standard 18-decimal precision before relaying actions to the `KernelManager` contract on the Skate chain. This ensures consistent mathematical operations within the core protocol.

Users initiate swaps by calling the `swap(...)` function in the `PeripheryPool` contract. This function accepts `amountSpecified` and an `extraData` parameter, which contains the `minAmountOutOrMaxAmountIn` value for slippage protection.

The `swap(...)` function correctly normalizes the `amountSpecified` (the input amount) to 18 decimals. However, it fails to normalize the `minAmountOutOrMaxAmountIn` value decoded from `extraData`. This unnormalized value is then passed along with the normalized input amount to the `KernelManager`.

```
// src/amm/periphery/PeripheryPool.sol
function swap(
    bytes32 recipient,
    bool zeroForOne,
    int256 amountSpecified,
    uint160 sqrtPriceLimitX96,
    bytes memory extraData
) external override {
    // ...
    // @audit minAmountOutOrMaxAmountIn is decoded but not normalized.
    (uint256 minAmountOutOrMaxAmountIn, uint256 destChainId, uint256 destVmType) =
        abi.decode(extraData, (uint256, uint256, uint256));

    // ...
    (uint256 amountIn, uint256 amountOut) = exactIn
        ? (uint256(amountSpecified), minAmountOutOrMaxAmountIn)
        : (minAmountOutOrMaxAmountIn, uint256(-amountSpecified));

    // ...
    // @audit `amountIn` is correctly normalized.
    uint256 amountInNormalized = _normalizeTokenAmount(amountIn, tokenIn);

    // ...
    action.kernelAppCalldata = abi.encodeWithSignature(
        "swap(bytes)",
        abi.encode(
            // ...
            int256(amountInNormalized),
            sqrtPriceLimitX96,
            // @audit `extraData` containing the unnormalized value is passed to the kernel.
            extraData
        )
    );
    // ...
}
```

The `KernelManagerLib` contract on the Skate chain decodes this data and performs its swap logic, including the slippage check. The check compares the calculated output amount (which is normalized to 18 decimals) against the unnormalized `minAmountOutOrMaxAmountIn` provided by the user.

```sol
// src/amm/kernel/libraries/KernelManagerLib.sol
function swap(
    DataTypes.State storage state,
    bytes calldata swapData
) external returns (IMessageBox.Task[] memory tasks) {
    // ...
    // @audit `extraData` is decoded again here. `minAmountOutOrMaxAmountIn` is not normalized.
    (uint256 minAmountOutOrMaxAmountIn,,) = abi.decode(extraData, (uint256, uint256, uint256));
    vars.exactIn = amountSpecified > 0;
    if (zeroForOne) {
        vars.amount0ToSettle =
            vars.exactIn ? uint256(amountSpecified) : minAmountOutOrMaxAmountIn;
        // @audit `vars.amount1ToTransfer` is a normalized value.
        vars.amount1ToTransfer = uint256(-vars.amount1Signed);
        // @audit-issue The following line may underflow.
        vars.amount0ToTransfer = vars.amount0ToSettle - uint256(vars.amount0Signed);
    } else {
        // @audit `vars.amount0ToTransfer` is a normalized value.
        vars.amount0ToTransfer = uint256(-vars.amount0Signed);
        vars.amount1ToSettle =
            vars.exactIn ? uint256(amountSpecified) : minAmountOutOrMaxAmountIn;
        // @audit-issue The following line may underflow.
        vars.amount1ToTransfer = vars.amount1ToSettle - uint256(vars.amount1Signed);
    }

    // @audit-issue Slippage check compares a normalized amount with an unnormalized amount.
    require(
        (zeroForOne ? vars.amount1ToTransfer : vars.amount0ToTransfer)
            >= (vars.exactIn ? minAmountOutOrMaxAmountIn : uint256(-amountSpecified)),
        "PeripheryPool::swap: slippage check"
    );
    // ...
}
```

This discrepancy has a severe impact when swapping tokens with decimals less than 18. For an exact-input swap, the calculated output amount (`amount1ToTransfer` or `amount0ToTransfer`) will be an 18-decimal normalized value, while `minAmountOutOrMaxAmountIn` will be a native, non-normalized value. The slippage check becomes `normalized_output >= unnormalized_minimum_output`, which will almost always pass, even if the user receives far fewer tokens than their minimum expectation. This allows a malicious actor or unfavorable market conditions to cause significant financial loss to the user.

Furthermore, for exact-output swaps, the logic uses the unnormalized `minAmountOutOrMaxAmountIn` (representing `maxAmountIn`) in calculations with normalized values. This can lead to arithmetic underflows and cause the transaction to revert.

Recommendation(s): Consider normalizing the `minAmountOutOrMaxAmountIn` parameter.

Status: Fixed

Update from the client: The remediation is applied. commit

6.6 [High] Incorrect liquidity recalculation leads to loss of funds

File(s): src/amm/kernel/libraries/KernelManagerLib.sol

Description: The `mint(...)` and `increaseLiquidity(...)` function in the `KernelManagerLib` contract is used to add liquidity to a pool. It internally calls the `_calcAmountsUsedInMintAndValidate(...)` helper function to determine the amount of liquidity to be minted and to perform slippage checks.

The `_calcAmountsUsedInMintAndValidate(...)` function first correctly calculates the `liquidityAmount` based on the user's desired deposit amounts. It then calls `pool.mint(...)`, which mints the liquidity and returns the actual token amounts used, `amount0Used` and `amount1Used`. However, the function then incorrectly recalculates the `liquidityAmount` using these returned amounts. This logic deviates from the standard Uniswap V3 implementation.

```
function _calcAmountsUsedInMintAndValidate(
    IKernelPool pool,
    uint160 sqrtPriceX96,
    int24 tickLower,
    int24 tickUpper,
    uint256 amount0,
    uint256 amount1,
    bytes memory extraData
)
    private
    returns (uint256 amount0Used, uint256 amount1Used, uint128 liquidityAmount)
{
    // @audit First, the correct liquidity is calculated from the desired amounts.
    liquidityAmount = LiquidityAmounts.getLiquidityForAmounts(
        sqrtPriceX96,
        tickLower.getSqrtRatioAtTick(),
        tickUpper.getSqrtRatioAtTick(),
        amount0,
        amount1
    );
    // @audit The pool mints the correct amount of liquidity.
    (amount0Used, amount1Used) =
        pool.mint(address(this), tickLower, tickUpper, liquidityAmount, "", false);

    // @audit-issue The liquidity amount is incorrectly recalculated here using the actual amounts used.
    // This will likely result in a smaller liquidity value than what was actually minted.
    liquidityAmount = LiquidityAmounts.getLiquidityForAmounts(
        sqrtPriceX96,
        tickLower.getSqrtRatioAtTick(),
        tickUpper.getSqrtRatioAtTick(),
        amount0Used,
        amount1Used
    );
    (uint256 amount0Min, uint256 amount1Min) = abi.decode(extraData, (uint256, uint256));
    require(amount0Used >= amount0Min && amount1Used >= amount1Min, "price slippage check");
}
```

The `mint(...)` and `increaseLiquidity(...)` function in `KernelManagerLib` then uses this incorrect, smaller `liquidityAmount` to create the user's NFT position.

```
function mint(
    DataTypes.State storage state,
    bytes calldata mintData
)
    external
    returns (IMessageBox.Task[] memory tasks)
{
    // ...
    (vars.amount0Used, vars.amount1Used, vars.liquidityAmount) =
    _calcAmountsUsedInMintAndValidate(
        vars.pool, vars.sqrtPriceX96, tickLower, tickUpper, amount0, amount1, extraData
    );
    // ...
    // @audit The incorrect, smaller liquidityAmount is stored in the user's NFT position.
    state.nftPositions[vars.tokenId] = DataTypes.NFTPosition({
        pool: kernelPool,
        tickLower: tickLower,
        tickUpper: tickUpper,
        liquidity: vars.liquidityAmount,
        // ...
    });
    // ...
}
```

Due to the internal logic of `LiquidityAmounts.getLiquidityForAmounts(...)`, the recalculated `liquidityAmount` will often be less than the amount of liquidity that was actually added to the pool. When this smaller value is stored with the user's NFT, their position is underrepresented. This will prevent the user from withdrawing their full share of the liquidity and collecting all the fees they are entitled to, leading to a direct and permanent loss of funds.

Recommendation(s): Consider removing the recalculation of `liquidityAmount` in the `_calcAmountsUsedInMintAndValidate(...)` function. The liquidity amount returned by the first call to `pool.mint(...)` is the correct amount that was minted and should be associated with the user's NFT position. The logic should be aligned with the reference Uniswap V3 implementation.

Status: Fixed

Update from the client: The remediation is applied. commit

6.7 [High] Ineffective Slippage Protection and Denial of Service for Exact-Output Swaps

File(s): solana/src/lib.rs

Description: The `swap(...)` function in the `periphery_pool` program on Solana allows users to initiate token swaps. It accepts a `min_amount_out_or_max_amount_in` parameter, which is crucial for protecting users from slippage. For exact-input swaps, this parameter specifies the minimum amount of output tokens the user is willing to accept. For exact-output swaps, it defines the maximum amount of input tokens the user is willing to spend. After a user calls `swap(...)`, the function constructs a `KernelAction::Swap` payload and sends it to the `action_box` program. An off-chain executor then relays this action to the Kernel on the Skate chain for final processing and settlement.

However, the `KernelAction::Swap` struct defined in `kernel_action.rs` does not include a field for the `min_amount_out_or_max_amount_in` parameter. This means this critical slippage protection value is not passed to the Kernel. The off-chain executor processing the action is forced to use a default value, which is `0`, when executing the swap logic on the Skate chain.

This leads to two significant issues:

1. **No Slippage Protection**: For exact-input swaps, the slippage check on the Kernel side becomes ineffective because it compares the actual output amount against a minimum of `0`. This exposes users to substantial financial loss, as their swaps can be executed at extremely unfavorable prices during volatile market conditions or by a malicious actor front-running their transaction;
2. **Denial of Service for Exact-Output Swaps**: For exact-output swaps, the Kernel logic uses the `max_amount_in` value (which it receives as `0`) in its calculations. This leads to an arithmetic underflow, causing the transaction on the Skate chain to revert. Consequently, all exact-output swaps initiated from the Solana periphery are non-functional, representing a denial of service for a core feature of the AMM;

```
// File: solana/periphery_pool/src/lib.rs

pub fn swap(
    ctx: Context<SwapCtx>,
    recipient: String,
    zero_for_one: bool,
    amount_specified: i64,
    sqrt_price_limit_x96: String,
    // @audit This parameter is used for local checks but is not included in the
    // kernel_action payload, leaving the kernel without slippage protection information.
    min_amount_out_or_max_amount_in: u64,
    dest_chain_id: u32,
    dest_vm_type: u8,
    _extra_data: Vec<u8>,
    action_box_seed: u64,
) -> Result<()> {
    // ...
    let (amount_in, amount_out) = if exact_in {
        (amount_specified as u64, min_amount_out_or_max_amount_in)
    } else {
        let amount_out_val = amount_specified.checked_neg().ok_or(PeripheryError::InvalidAmounts)?;
        (min_amount_out_or_max_amount_in, amount_out_val as u64)
    };
    // ...
    // @audit-issue The `KernelAction` does not include `min_amount_out_or_max_amount_in`.
    let kernel_action = KernelAction::Swap {
        action_id: action_id.clone(),
        chain_id: pool.chain_id,
        src_vm_type: pool.src_vm_type,
        dest_chain_id,
        dest_vm_type,
        recipient: recipient.clone(),
        user: user_bytes32.clone(),
        kernel_pool: pool.kernel_pool.clone(),
        zero_for_one,
        amount_specified,
        sqrt_price_limit_x96: sqrt_price_limit_x96.clone(),
        extra_data: _extra_data.clone(),
    };
    // ...
}
```

```solidity
// File: src/amm/kernel/libraries/KernelManagerLib.sol

    function swap(
        DataTypes.State storage state,
        bytes calldata swapData
    )
        external
        returns (IMessageBox.Task[] memory tasks)
    {
        // ...
        SwapLocalVars memory vars;
        (vars.amount0Signed, vars.amount1Signed, vars.sqrtPriceX96, vars.liquidity, vars.tick) =
        IKernelPool(kernelPool).swap(
            _recipient, zeroForOne, amountSpecified, sqrtPriceLimitX96, extraData, false
        );

        (uint256 minAmountOutOrMaxAmountIn,,) = abi.decode(extraData, (uint256, uint256, uint256));
        vars.exactIn = amountSpecified > 0;
        if (zeroForOne) {
            // @audit It will be 0 if exactIn is false
            vars.amount0ToSettle =
                vars.exactIn ? uint256(amountSpecified) : minAmountOutOrMaxAmountIn;
            vars.amount1ToTransfer = uint256(-vars.amount1Signed);
            // @audit-issue It will revert due to underflow as vars.amount0ToSettle will be 0 if exactIn is false
            vars.amount0ToTransfer = vars.amount0ToSettle - uint256(vars.amount0Signed);
        } else {
            vars.amount0ToTransfer = uint256(-vars.amount0Signed);
            // @audit It will be 0 if exactIn is false
            vars.amount1ToSettle =
                vars.exactIn ? uint256(amountSpecified) : minAmountOutOrMaxAmountIn;
            // @audit-issue It will revert due to underflow as vars.amount1ToSettle will be 0 if exactIn is false
            vars.amount1ToTransfer = vars.amount1ToSettle - uint256(vars.amount1Signed);
        }

        // ...
    }
```

Recommendation(s): Consider revisiting the cross-chain communication mechanism for swap operations. The current implementation fails to propagate the user's slippage preferences from the Solana periphery to the Kernel. A revised design should ensure that all critical parameters, including `min_amount_out_or_max_amount_in`, are reliably transmitted and enforced during execution on the Skate chain. This will restore the functionality of exact-output swaps and re-enable essential slippage protection for users performing exact-input swaps.

Status: Fixed

Update from the client: solana commit

6.8 [High] Rounding in favor of the user in `normalize_amount_for_transfer` leads to periphery pool insolvency

File(s): solana/src/helper.rs

Description: When a user interacts with periphery pool operations such as `mint(...)` or `increase_liquidity(...)`, they provide tokens to the contract. The amounts for these operations are standardized to a 6-decimal precision from the front-end. The `normalize_amount_for_transfer(...)` helper function is responsible for converting these standardized amounts to the token's native decimal precision before the actual `transfer_checked(...)` call is made to pull funds from the user.

However, for tokens with fewer than 6 decimals, the normalization logic involves division, which truncates the result. This effectively rounds the amount down in favor of the user. As a result, the actual number of tokens transferred from the user's wallet is less than the amount used for the pool's internal accounting and for the AMM calculations performed by the `KernelManager` on the Skate chain.

This discrepancy between the accounted amount and the actually transferred amount leads to a state where the protocol's internal ledger is inconsistent with its real token holdings. Over time, even during normal usage, these small rounding errors will accumulate across numerous transactions, causing the `PeripheryPool`'s accounted value to diverge from its actual reserves, eventually leading to insolvency. In a more severe scenario, a malicious actor could repeatedly exploit this vulnerability. By making a series of deposits and withdrawals, the attacker can consistently extract slightly more assets than they provided, systematically draining the `PeripheryPool` of funds.

```
pub fn normalize_amount_for_transfer(amount: u64, token_decimals: u8) -> Result<u64> {
    // ...
    else {
        // Need to divide the amount (e.g., token with 3 decimals)
        let decimal_diff = NORMALIZED_DECIMALS - token_decimals;
        let divisor = 10u64.pow(decimal_diff as u32);

        // @audit-issue Division truncates the result, rounding down the amount
        // transferred from the user, while accounting uses the original, larger amount.
        let adjusted_amount: u64 = amount.checked_div(divisor)
            .ok_or(PeripheryError::DivisionByZero)?;

        return Ok(adjusted_amount);
    }
}
```

Recommendation(s): Consider adjusting the rounding direction in the `normalize_amount_for_transfer(...)` function. When converting amounts for tokens with fewer decimals than the normalized standard, the calculation should round up to ensure the protocol always receives at least the amount it accounts for. This prevents discrepancies that could lead to the drainage of funds.

Status: Fixed

Update from the client: Fixed in solana commit

6.9 [High] Swap logic incorrectly handles exact output swaps leading to failed slippage protection

File(s): src/amm/periphery/PeripheryPool.sol, src/amm/kernel/libraries/KernelManagerLib.sol

Description: The swap functionality in the Skate AMM is designed to support both exact input and exact output swaps, similar to Uniswap V3. A user specifies an exact input swap with a positive `amountSpecified` and an exact output swap with a negative `amountSpecified`. However, the implementation incorrectly processes exact output swaps, leading to failed slippage protection and users swapping more tokens than intended.

The issue begins in the `PeripheryPool.swap(...)` function. When a user provides a negative `amountSpecified` for an exact output swap, the `exactIn` flag is correctly identified as `false`. The `amountIn` variable is set to `minAmountOutOrMaxAmountIn`, which represents the maximum input the user is willing to spend.

```
// src/amm/periphery/PeripheryPool.sol
function swap(
    // ...
)
    external
    override
{
    // ...
    bool exactIn = amountSpecified > 0;
    (uint256 amountIn, uint256 amountOut) = exactIn
        ? (uint256(amountSpecified), minAmountOutOrMaxAmountIn)
        : (minAmountOutOrMaxAmountIn, uint256(-amountSpecified));
    // ...
}
```

However, when creating the action to be sent to the kernel, the `amountSpecified` field in the `kernelAppCalldata` is encoded as `int256(amountInNormalized)`. Since `amountIn` is always a positive value, the `amountSpecified` sent to the kernel will always be positive.

```
// src/amm/periphery/PeripheryPool.sol
// ...
action.kernelAppCalldata = abi.encodeWithSignature(
    "swap(bytes)",
    abi.encode(
        // ...
        // @audit-issue The `amountSpecified` sent to the kernel is derived from `amountIn`,
        // which is always positive. This erases the distinction between the
        // exact-input and exact-output swaps.
        int256(amountInNormalized),
        sqrtPriceLimitX96,
        extraData
    )
);
// ...
```

Consequently, the `KernelManagerLib.swap(...)` function always receives a positive `amountSpecified`, causing it to interpret every swap as an exact input swap. This breaks the logic for handling exact output swaps.

The primary impact is the failure of the slippage protection mechanism for what should be exact output swaps. In `KernelManagerLib.swap(...)`, the `vars.exactIn` flag is always `true`. The slippage check is as follows:

```
// src/amm/kernel/libraries/KernelManagerLib.sol
// ...
require(
    // @audit `vars.exactIn` is always true, so this check is performed with minAmountOutOrMaxAmountIn.
    (zeroForOne ? vars.amount1ToTransfer : vars.amount0ToTransfer)
        >= (vars.exactIn ? minAmountOutOrMaxAmountIn : uint256(-amountSpecified)),
    "PeripheryPool::swap: slippage check"
);
// ...
```

For an intended exact output swap, `minAmountOutOrMaxAmountIn` represents the user's _maximum input amount_. The check incorrectly compares the actual output amount (`vars.amount1ToTransfer` or `vars.amount0ToTransfer`) against this maximum input amount (`amountOut >= maxAmountIn`). This comparison is misleading and does not offer effective slippage protection. The correct check should ensure the actual input amount does not exceed the user's specified maximum.

This flaw can cause transactions to revert incorrectly or, more critically, allow swaps to execute with significant negative slippage, causing financial loss for the user. Furthermore, since the swap is always treated as exact input using the user's `maxAmountIn`, the user may swap a larger amount of tokens than necessary to achieve their desired output amount.

Recommendation(s): Consider revisiting the approach for how swap parameters are passed from the periphery to the kernel. One potential path is to forward the original, user-provided `amountSpecified` to the kernel layer, ensuring its sign is preserved after normalization, rather than calculating a new positive value from the `amountIn`. Following this or any alternative change, it is advisable to perform a holistic review of the entire swap-related logic across all contracts to ensure the necessary adjustments are made for end-to-end consistency and correctness.

Status: Fixed

Update from the client: The remediation is applied. commit

6.10 [High] The `initialize_tokens` instruction lacks access control and validation

File(s): solana/src/lib.rs

Description: The `PeripheryPool` contract is designed to be initialized in two steps. First, the `initialize_pool(...)` instruction creates the `PeripheryPool` state account, setting its configuration, including the `manager` address. Second, the `initialize_tokens(...)` instruction is called to create the token accounts (`pool_token0` and `pool_token1`) that will hold the pool's assets. This second instruction is intended to be called by the `manager`.

However, the `InitializeTokens` accounts context is missing a crucial access control check. It does not verify that the `manager` signer is the same one stored in the `periphery_pool` account. This allows anyone to call `initialize_tokens(...)` for any existing pool.

Furthermore, the instruction can be called multiple times for the same pool with different token mints. Because the pool's token accounts are created as PDAs using the token mint's address as a seed (`seeds = [b"pool-token", periphery_pool.key().as_ref(), token0.key().as_ref()]`), an attacker can provide a different mint address to create a new, distinct token account PDA, bypassing the `init` constraint that would otherwise prevent re-initialization.

This can lead to a denial-of-service condition. An attacker can call `initialize_tokens(...)` on a legitimate, existing pool and provide malicious or incorrect token mints. This will re-initialize the `token0` and `token1` pubkeys stored in the `PeripheryPool` state. When off-chain services later attempt to settle actions (e.g., `settle_mint(...)`), their transactions will fail because the token accounts they derive will not match the new, incorrect ones on-chain. This effectively bricks the periphery pool, halting all operations. The protocol cannot fix the legitimate pools by calling `initialize_tokens(...)` again with the correct values which were previously set, because the token account PDAs were already created, causing any subsequent `init` attempt to revert.

```
#[derive(Accounts)]
pub struct InitializeTokens<'info> {
    // @audit-issue Access control is missing. Anyone can call this instruction.
    #[account(mut)]
    pub periphery_pool: Box<Account<'info, PeripheryPool>>,
    #[account(
        seeds = [b"pool-authority", periphery_pool.key().as_ref()],
        bump,
    )]
    /// CHECK: This is a derived PDA used as token account authority.
    pub pool_authority: UncheckedAccount<'info>,
    // @audit Using a different token mint will result in a new PDA,
    // allowing re-initialization of the pool's token accounts.
   #[account(
        init,
        payer = manager,
        token::mint = token0,
        token::authority = pool_authority,
        token::token_program = token_program0,
        seeds = [b"pool-token", periphery_pool.key().as_ref(), token0.key().as_ref()],
        bump,
    )]
    pub pool_token0: Box<InterfaceAccount<'info, TokenAccount>>,
    #[account(
        init,
        payer = manager,
        token::mint = token1,
        token::authority = pool_authority,
        token::token_program = token_program1,
        seeds = [b"pool-token", periphery_pool.key().as_ref(), token1.key().as_ref()],
        bump,
    )]
    pub pool_token1: Box<InterfaceAccount<'info, TokenAccount>>,
    // ...
}
```

Recommendation(s): Consider implementing the following mitigations:

- Enforce strict access control on the `initialize_tokens(...)` instruction. The instruction should only be executable by the `manager` address that was set during the pool's creation and is stored within the `PeripheryPool` state;
- Introduce a state check to prevent re-initialization. The logic should ensure that the token accounts for a `PeripheryPool` can only be set once, preventing any party from overwriting the correct token addresses;
- Validate the `PeripheryPool` account by deriving its address from its creation seeds within the `initialize_tokens(...)` instruction. This ensures that the `token0` and `token1` mints used for token account creation are the same ones used when the pool itself was created, maintaining data integrity;

Status: Fixed

Update from the client: Fixed in solana commit

6.11 [Medium] Unsafe usage of ERC-20 token transfer functions

File(s): src/amm/periphery/PeripheryPool.sol

Description: The PeripheryPool contract facilitates Automated Market Maker (AMM) operations, which involve frequent transfers of ERC-20 tokens to and from users. These transfers are performed using the standard transfer(...) and transferFrom(...) functions.

However, the contract does not check the boolean return value of these ERC-20 token transfer calls. According to the ERC-20 standard, these functions can return false to signal a failed transfer without reverting the transaction. Certain tokens, such as those with fee-on-transfer mechanisms, pausable contracts, or blacklisting features, may utilize this return pattern.

By not verifying the return value, the contract will continue its execution as if the transfer was successful, even when it failed. This creates a discrepancy between the contract's internal state (e.g., updating a user's deposit amounts in usersData) and the actual token balances. This can lead to incorrect accounting and potential loss of funds for either the user or the protocol.

The issue is present in multiple functions, as seen in the mint(...) function:

```
function mint(...) external override {
    // ...
    UserData storage data = usersData[msg.sender];
    data.amount0 += amount0;
    data.amount1 += amount1;
    // ...
    if (amount0 != 0) {
        // @audit-issue The return value of `transferFrom` is not checked.
        IERC20(token0).transferFrom(msg.sender, address(this), amount0);
    }
    if (amount1 != 0) {
        // @audit-issue The return value of `transferFrom` is not checked.
        IERC20(token1).transferFrom(msg.sender, address(this), amount1);
    }
    // ...
}
```

This vulnerability pattern is repeated across several functions in the PeripheryPool contract that handle token movements. The following is a list of affected functions:

- mint(...);
- increaseLiquidity(...);
- removeStagedAssets(...);
- swap(...);
- _transferTo(...), which is used by all settlement functions;

A silent failure in any of these transfers would cause the protocol's logic to diverge from the reality of token ownership, potentially leading to accounting errors and value loss.

Recommendation(s): Consider using OpenZeppelin's SafeERC20 library for all ERC-20 token operations to ensure that token transfers are always executed successfully or the entire transaction is rolled back, maintaining state consistency.

Status: Fixed

Update from the client: The remediation is applied. commit

6.12 [Low] Dust amounts will get stuck in `PeripheryPool` contracts

File(s): src/amm/periphery/PeripheryPool.sol, solana/src/lib.rs

Description: The Skate AMM architecture involves `KernelPool` contracts on the Skate chain that handle all core logic, and `PeripheryPool` contracts on various chains that hold user funds and interact with the kernel. A key design choice is that the `KernelPool` operates with token amounts that are normalized to 18 decimals for its internal calculations, regardless of the token's actual decimals.

The `PeripheryPool` contract is responsible for converting token amounts between their native decimal representation and the normalized 18-decimal representation used by the kernel. This is done via the `_normalizeTokenAmount(...)` and `_denormalizeTokenAmount(...)` functions.

The `_denormalizeTokenAmount(...)` function converts a normalized 18-decimal amount back to the token's native decimal amount by dividing by `10 ** (18 - tokenDecimals)`.

```
function _denormalizeTokenAmount(
    uint256 amount,
    address token
)
    private
    view
    returns (uint256)
{
    // @audit-issue Precision is lost here due to integer division.
    return amount / 10 ** (NORMALIZED_DECIMAL - IERC20Metadata(token).decimals());
}
```

This function is used in `_transferTo(...)` before transferring funds out to users.

```
function _transferTo(address user, uint256 amount0, uint256 amount1) internal {
    uint256 amount0Denormalized = _denormalizeTokenAmount(amount0, token0);
    uint256 amount1Denormalized = _denormalizeTokenAmount(amount1, token1);
    (uint256 amount0Available, uint256 amount1Available) = balancesAvailable();
    // ...
    if (amount0Denormalized != 0) IERC20(token0).transfer(user, amount0Denormalized);
    if (amount1Denormalized != 0) IERC20(token1).transfer(user, amount1Denormalized);
    // ...
}
```

The problem is that the integer division in `_denormalizeTokenAmount(...)` truncates any remainder. For tokens with fewer than 18 decimals, any normalized amount calculated by the `KernelPool` that is less than `10 ** (18 - tokenDecimals)` will be rounded down to zero when denormalized. The `KernelPool` performs high-precision calculations for swap outputs, fees, and liquidity provider shares. When these precise, normalized amounts are sent to the `PeripheryPool` to be paid out, the truncated remainder (dust) is not transferred. However, this dust amount was accounted for in the `KernelPool`'s state. As a result, these small amounts of tokens will accumulate in the `PeripheryPool` contract over time.

Since there is no function to withdraw these residual dust balances, they will be permanently locked in the contract, leading to a gradual but irreversible loss of funds for the protocol's users.

The same issue is present in the Solana's Periphery Pool program.

Recommendation(s): Consider implementing a mechanism to track the accumulated dust amounts. A privileged function could be added to allow a manager or owner role to periodically sweep these dust funds to a designated treasury or DAO address. This would prevent the funds from being permanently locked and allow for their recovery.

Status: Fixed

Update from the client: The remediation is applied. commit

Update from the Nethermind Security: The issue is still present in the Solana's Periphery Pool program.

Update from the client: Fixed in https://github.com/Skate-Org/Skate-AMM/commit/bb8f07fd1d0c388391382f42c6c9511790a72db4

6.13 [Low] Exact output swaps may revert unexpectedly when a price limit is reached

File(s): src/amm/kernel/libraries/KernelManagerLib.sol

Description: The Skate AMM is designed to mirror the core mathematical logic of Uniswap V3. In an "exact output" swap, a user specifies the precise amount of tokens they wish to receive (`amountOut`) and a maximum amount of input tokens they are willing to pay. Users can also set a price limit (`sqrtPriceLimitX96`) to protect against unfavorable price movements.

In the standard Uniswap V3 implementation, if an exact output swap hits the specified price limit before the full `amountOut` has been acquired, the swap is not reverted. Instead, it completes with the amount of output tokens that could be obtained before the price limit was reached. The actual amount received will be less than the amount requested.

However, the `swap(...)` function in the `KernelManagerLib` contract enforces a stricter requirement. After the underlying `KernelPool.swap(...)` call is completed, it checks if the actual amount of output tokens is greater than or equal to the amount the user originally requested.

```
// src/amm/kernel/libraries/KernelManagerLib.sol
function swap(
    DataTypes.State storage state,
    bytes calldata swapData
)
    external
    returns (IMessageBox.Task[] memory tasks)
{
    // ...
    (vars.amount0Signed, vars.amount1Signed, vars.sqrtPriceX96, vars.liquidity, vars.tick) =
    IKernelPool(kernelPool).swap(
        _recipient, zeroForOne, amountSpecified, sqrtPriceLimitX96, extraData, false
    );

    (uint256 minAmountOutOrMaxAmountIn,,) = abi.decode(extraData, (uint256, uint256, uint256));
    vars.exactIn = amountSpecified > 0;
    if (zeroForOne) {
        vars.amount0ToSettle =
            vars.exactIn ? uint256(amountSpecified) : minAmountOutOrMaxAmountIn;
        vars.amount1ToTransfer = uint256(-vars.amount1Signed);
        vars.amount0ToTransfer = vars.amount0ToSettle - uint256(vars.amount0Signed);
    } else {
        vars.amount0ToTransfer = uint256(-vars.amount0Signed);
        vars.amount1ToSettle =
            vars.exactIn ? uint256(amountSpecified) : minAmountOutOrMaxAmountIn;
        vars.amount1ToTransfer = vars.amount1ToSettle - uint256(vars.amount1Signed);
    }

    // @audit-issue This check is stricter than Uniswap V3's.
    require(
        (zeroForOne ? vars.amount1ToTransfer : vars.amount0ToTransfer)
            >= (vars.exactIn ? minAmountOutOrMaxAmountIn : uint256(-amountSpecified)),
        "PeripheryPool::swap: slippage check"
    );
    // ...
}
```

Because the underlying `KernelPool` inherits Uniswap V3's behavior, it can return an actual output amount (`uint256(-vars.amount1Signed)` or `uint256(-vars.amount0Signed)`) that is less than the requested amount (`uint256(-amountSpecified)`) when a price limit is hit. In such cases, the `require` statement in `KernelManagerLib` will fail, causing the entire transaction to revert.

This behavior is inconsistent with the established Uniswap V3 pattern, where such a swap would partially succeed. This can lead to failed transactions and a degraded user experience, particularly during periods of high market volatility when price limits are more likely to be triggered.

Recommendation(s): Consider revisiting the logic that handles exact output swaps. The current implementation enforces that the entire requested output amount must be fulfilled, which causes swaps to revert if a price limit is hit prematurely. This behavior is inconsistent with the underlying Uniswap V3 protocol, which would allow for a partial fulfillment in such scenarios. Re-evaluating this strict requirement could prevent unexpected transaction failures and provide a more consistent and user-friendly experience, especially in volatile market conditions.

Status: Fixed

Update from the client: The check was removed as part of the finding "Swap behavior deviates from Uniswap V3 regarding sqrtPriceLimitX96" and the correct check against "maxInputAmount" is introduced.

6.14 [Info] Fee Collection Mechanism Lacks Granularity

File(s): src/amm/periphery/PeripheryPool.sol, src/amm/kernel/libraries/KernelManagerLib.sol, solana/src/lib.rs

Description: In the Uniswap V3 protocol, the `collect(...)` function allows users to specify `amount0Max` and `amount1Max`, providing granular control over fee withdrawals. This enables users to perform partial fee collections, such as withdrawing only one of the two token fees or claiming a specific amount.

The Skate protocol consolidates fee collection into the `decreaseLiquidity(...)` function, which is triggered with a `liquidityAmount` of `0`. When this function is used for fee collection, the underlying `burn(...)` call returns zero for both token amounts. This design requires the user to pass `amount0Min = 0` and `amount1Min = 0` to satisfy the slippage check.

Following this, the logic in `KernelManagerLib.decreaseLiquidity(...)` proceeds to collect the _entirety_ of the fees owed to the position.

```
// src/amm/kernel/libraries/KernelManagerLib.sol
function decreaseLiquidity(
    DataTypes.State storage state,
    bytes calldata decreaseLiquidityData
)
    // ...
{
    // ...
    // @audit When liquidityAmount is 0, vars.amount0 and vars.amount1 will be 0.
    (vars.amount0, vars.amount1) = vars.pool.burn(
        address(this), position.tickLower, position.tickUpper, liquidityAmount, false
    );
    (vars.amount0Min, vars.amount1Min) = abi.decode(extraData, (uint256, uint256));

    // @audit This check forces amount{0,1}Min to be 0 when collecting fees.
    require(
        vars.amount0 >= vars.amount0Min && vars.amount1 >= vars.amount1Min,
        "price slippage check"
    );

    // @audit-issue The subsequent collect call withdraws all available fees, offering no user control.
    (vars.amount0Collected, vars.amount1Collected) = vars.pool.collect(
        address(this),
        position.tickLower,
        position.tickUpper,
        position.tokensOwed0,
        position.tokensOwed1
    );
    // ...
}
```

This implementation results in an "all-or-nothing" fee withdrawal mechanism. Users do not have the ability to perform partial fee collections. Every call to collect fees withdraws the full amount of both tokens owed. This is a functional difference from the Uniswap V3 protocol, which offers more flexibility in managing accrued fees.

The same issue is present in the Solana's Periphery Pool program.

Recommendation(s): Consider revisiting the fee collection design to provide users with more granular control over withdrawing their accrued fees. This would better align with the flexibility offered by the original Uniswap V3 protocol and enhance the user experience for position managers.

Status: Acknowledged

Update from the client: To keep the withdrawal simpler in cross-chain environment we will keep the current approach and consider incorporting granularity in the next versions.

6.15 [Info] Incorrect accounting of global staged assets in `swap(...)`

File(s): solana/src/lib.rs

Description: The `PeripheryPool` contract maintains two levels of accounting for assets staged by users for various actions: a per-user accounting within the `UserDataAccount` PDA, and a global accounting within the `PeripheryPool` account state itself (`staged_token0_amount` and `staged_token1_amount`).

Functions like `mint(...)` and `increase_liquidity(...)` correctly increment both the per-user and the global staged asset counters.

However, the `swap(...)` function only updates the per-user `UserDataAccount` and fails to increment the global `staged_token0_amount` or `staged_token1_amount` counters in the `PeripheryPool`.

```
pub fn swap(
    // ...
) -> Result<()> {
    // ...
    let user_data: &mut Account<'_, UserDataAccount> = &mut ctx.accounts.user_data;
    // @audit-issue Global staged asset counters are NOT incremented here.
    if zero_for_one {
        // @audit Only the per-user amount is incremented.
        user_data.amount0 = user_data
            .amount0
            .checked_add(amount_in)
            .ok_or(PeripheryError::Overflow)?;
    } else {
        // @audit Only the per-user amount is incremented.
        user_data.amount1 = user_data
            .amount1
            .checked_add(amount_in)
            .ok_or(PeripheryError::Overflow)?;
    }
    // ...
}
```

The `remove_staged_assets(...)` function, which is intended to be called when an action fails one the kernel side, decrements both the per-user and global counters. If a swap action fails and this function is called, it will attempt to decrement the global counters by an amount they were never incremented by. This will cause an underflow. The issue is masked because the code uses `.unwrap_or(0)`, which prevents a panic but sets the global counter to `0`, leading to an incorrect state.

```
pub fn remove_staged_assets(ctx: Context<RemoveStagedAssetsCtx>) -> Result<()> {
    // ...
    let amount0 = user_data.amount0;
    let amount1 = user_data.amount1;
    // ...
    let pool = &mut ctx.accounts.periphery_pool;
    // @audit-issue This will underflow if a swap fails, but `unwrap_or(0)` masks the error.
    pool.staged_token0_amount = pool.staged_token0_amount.checked_sub(amount0).unwrap_or(0);
    // @audit-issue This will underflow if a swap fails, but `unwrap_or(0)` masks the error.
    pool.staged_token1_amount = pool.staged_token1_amount.checked_sub(amount1).unwrap_or(0);
    // ...
}
```

While this logic discrepancy does not cause a direct loss of funds or a denial-of-service due to the `unwrap_or(0)` pattern, it results in inconsistent state within the contract. Any off-chain services or monitoring tools that rely on the `staged_token0_amount` and `staged_token1_amount` values from the `PeripheryPool` account will read incorrect data.

Recommendation(s): Consider either removing the global staged asset counters (`staged_token0_amount` and `staged_token1_amount`) if they are redundant, or ensure they are consistently updated in the `swap(...)` function to maintain correct and reliable accounting across the protocol.

Status: Fixed

Update from the client: Fixed in solana commit

6.16 [Info] Missing Uniswap V3 standard parameters like `deadline` and `recipient`

File(s): `src/amm/periphery/PeripheryPool.sol`, `solana/src/lib.rs`

Description: The `PeripheryPool` contract is the primary entry point for users to interact with the Skate AMM on periphery chains. It exposes functions for managing liquidity (`mint(...)`, `increaseLiquidity(...)`, `burn(...)`, `decreaseLiquidity(...)`) and performing swaps (`swap(...)`). These functions deviate from the standard Uniswap V3 implementation by omitting key parameters that provide user protection and composability.

Crucially, all of these price-sensitive functions lack a `deadline` parameter. In Uniswap V3, this parameter is a standard safety feature to protect users from their transactions being withheld by miners or MEV bots and executed much later at a significantly worse price. This omission exposes users to potential financial losses due to slippage from delayed execution.

Furthermore, the `mint(...)` function also omits the `recipient` parameter, which is present in Uniswap V3's implementation. This forces the newly minted position NFT to be sent to `msg.sender`, reducing the protocol's composability by preventing contracts or users from minting positions on behalf of others.

The same problem is present in the Solana's Periphery Pool program.

Recommendation(s): Consider aligning the function signatures in `PeripheryPool` with the Uniswap V3 standards to enhance user security and protocol composability.

Status: Fixed

Update from the client: The remediation is applied. commit and solana commit

Update from the Nethermind Security: The `burn(...)` and `burn_legacy(...)` instructions in the Solana's Periphery Pool program is missing the `deadline` parameter.

Update from the client: Fixed in `bb8f07f`.

6.17 [Info] Only NFT position owner can increase liquidity

File(s): src/amm/kernel/libraries/KernelManagerLib.sol

Description: The `KernelManager` contract facilitates the management of liquidity positions, which are represented as NFTs. The `increaseLiquidity(...)` function allows users to add more funds to their existing positions.

However, the current implementation in the `KernelManagerLib` contract restricts this action exclusively to the owner of the position NFT. The function verifies that the address initiating the transaction is the owner of the `tokenId` corresponding to the liquidity position.

```
function increaseLiquidity(
    DataTypes.State storage state,
    bytes calldata increaseLiquidityData
)
    external
    returns (IMessageBox.Task[] memory tasks)
{
    // ...
    (
        // ...
        bytes32 user,
        // ...
        uint256 tokenId,
        // ...
    ) = abi.decode(
        increaseLiquidityData,
        (bytes32, uint256, uint256, bytes32, address, uint256, uint256, uint256, bytes)
    );
    IncreaseLiquidityLocalVars memory vars;
    vars.user = _fromBytes32ToAddress(user);
    // @audit-issue This check requires the caller to be the owner of the NFT position.
    require(
        IKernelManager(address(this)).ownerOf(tokenId) == vars.user,
        "KernelManagerLib::increaseLiquidity: not owner of tokenId"
    );
    // ...
}
```

This design choice differs from other similar AMM protocols where anyone can "donate" or add liquidity to any existing position. This restriction can be limiting in several scenarios. For instance, a user who holds their position NFT in a secure cold wallet cannot add liquidity using a more convenient hot wallet. Furthermore, it prevents the development of third-party services or integrations that might want to top up user positions on their behalf.

Recommendation(s): Consider re-evaluating whether this ownership check is a strict product requirement or a precautionary measure. Allowing any address to increase the liquidity of a position could provide greater flexibility for both users and external integrators, enhancing the composability of the protocol.

Status: Fixed

Update from the client: The remediation is applied. commit

6.18 [Info] Potential `actionId` collision in `collectProtocol(...)`

File(s): src/amm/kernel/libraries/KernelManagerLib.sol

Description: The `collectProtocol(...)` function in the `KernelManagerLib` library is used by the `KernelManager` contract owner to withdraw accrued protocol fees. This process generates a task with a corresponding `actionId` that is intended to be unique for off-chain processing.

The `actionId` is calculated by hashing a combination of a fixed string ("collectProtocol"), the `recipient`, the `kernelPool` address, and the current `block.number`.

```
// src/amm/kernel/libraries/KernelManagerLib.sol
// ...
    tasks[0] = IMessageBox.Task({
        // ...
        actionId: keccak256(
            // @audit-issue The hash only depends on recipient, kernelPool, and block.number.
            abi.encodePacked("collectProtocol", recipient, kernelPool, block.number)
        )
    });
}
```

However, this hashing mechanism does not guarantee uniqueness if the function is called multiple times within the same block with the same `recipient` and `kernelPool` parameters. In such a scenario, all generated tasks will have an identical `actionId`.

This hash collision could lead to unexpected behavior in off-chain systems that rely on the uniqueness of `actionId` for task tracking and execution. For instance, an off-chain service might process the first task and ignore subsequent ones with the same ID, causing the owner's later fee collection attempts within the same block to be missed.

Recommendation(s): Consider incorporating a nonce or another source of entropy into the `actionId` calculation to ensure its uniqueness across multiple calls within the same block.

Status: Mitigated

Update from the client: The remediation is applied. commit

6.19 [Info] Refunded tokens from swaps are sent to the recipient address

File(s): src/amm/periphery/PeripheryPool.sol, src/amm/kernel/libraries/KernelManagerLib.sol, solana/src/lib.rs

Description: In the Skate AMM, users can initiate a swap by calling the `swap(...)` function on a `PeripheryPool` contract. This function allows specifying a `recipient` address that will receive the output tokens from the trade.

Unlike typical Uniswap V3-style swaps where only the exact required amount of input tokens is pulled from the user, the Skate AMM pulls the maximum specified input amount (the `amountSpecified` for exact-in swaps, or the maximum input amount from `extraData` for exact-out swaps) and stages it. After the swap is processed by the `KernelManager` on the Skate chain, any unused portion of the input token is refunded.

The issue is that these refunded input tokens are sent to the `recipient` of the swap, not to the original user who initiated the transaction and provided the funds. The calculation of the refunded amount and the swapped amount happens within the `swap(...)` function in the `KernelManagerLib` contract.

```
// src/amm/kernel/libraries/KernelManagerLib.sol
function swap(...) external returns (IMessageBox.Task[] memory tasks) {
    // ...
    if (zeroForOne) {
        // @audit The total input amount from the user.
        vars.amount0ToSettle =
            vars.exactIn ? uint256(amountSpecified) : minAmountOutOrMaxAmountIn;
        // @audit The amount of token1 to transfer to the recipient (swapped amount).
        vars.amount1ToTransfer = uint256(-vars.amount1Signed);
        // @audit The amount of token0 to transfer back (refunded amount).
        vars.amount0ToTransfer = vars.amount0ToSettle - uint256(vars.amount0Signed);
    } // ... else block for oneForZero swap has similar logic.

    // ...
    tasks = new IMessageBox.Task[](1);
    tasks[0] = IMessageBox.Task({
        appAddress: appAddress,
        taskCalldata: abi.encodeWithSignature(
            "settleSwap(bytes32,bytes32,uint256,uint256,uint256,uint256)",
            recipient,
            user,
            vars.amount0ToTransfer, // @audit This includes the refunded amount.
            vars.amount1ToTransfer, // @audit This includes the swapped amount.
            vars.amount0ToSettle,
            vars.amount1ToSettle
        ),
        // ...
    });
}
```

The `settleSwap(...)` function in the `PeripheryPool` contract then calls `_transferTo(...)`, which sends both the swapped tokens and the refunded input tokens to the `recipient`.

```
// src/amm/periphery/PeripheryPool.sol
function settleSwap(
    bytes32 recipient,
    bytes32 user, // @audit-issue Unused in the transfer logic.
    uint256 amount0ToTransfer,
    uint256 amount1ToTransfer,
    // ...
)
    external
    override
    onlyGateway
{
    // ...
    // @audit-issue The `recipient` receives both the swapped tokens and any refund.
    _transferTo(_fromBytes32ToAddress(recipient), amount0ToTransfer, amount1ToTransfer);
}
```

This behavior is counterintuitive. A user might specify a recipient contract for a swap, expecting it to only receive the output token. However, the contract will also receive the refunded input tokens. If the recipient contract is not designed to handle these unexpected tokens, the funds could become stuck, leading to a loss for the user.

The same issue is present in the Solana's Periphery Pool program.

Recommendation(s): Consider documenting this behavior so that users and developers are aware that the `recipient` address will receive both the output of the swap and any refunded input tokens. Alternatively, consider modifying the logic to send the refunded tokens back to

the account that originally provided the tokens for the swap. If cross-chain swaps are implemented in the future, this behavior might need to be revisited depending on the chosen implementation.

Status: Fixed

Update from the client: The remediation is applied. commit and solana commit

6.20 [Info] Swap behavior deviates from Uniswap V3 regarding `sqrtPriceLimitX96`

File(s): src/amm/periphery/PeripheryPool.sol, src/amm/kernel/libraries/KernelManagerLib.sol

Description: Users can initiate a token swap by calling the `swap(...)` function on the `PeripheryPool` contract. This action is then relayed to the `KernelManager`, which processes the intent via the `KernelManagerLib.swap(...)` library function. Finally, `KernelManagerLib` calls the `swap(...)` function on the corresponding `KernelPool` to execute the swap on the Skate chain.

The `KernelPool.swap(...)` function, which contains the core swap logic, requires a `sqrtPriceLimitX96` parameter to protect against slippage. This parameter must be a valid, non-zero value that is on the correct side of the current price, otherwise the transaction will revert.

The issue is that the periphery layer of the Skate AMM (`PeripheryPool` and `KernelManagerLib`) does not handle the `sqrtPriceLimitX96` parameter in the same user-friendly manner as Uniswap V3's `SwapRouter`. Specifically, it's passed directly to the `KernelPool`.

```
// src/amm/kernel/libraries/KernelManagerLib.sol
function swap(
    DataTypes.State storage state,
    bytes calldata swapData
) external returns (IMessageBox.Task[] memory tasks) {
    // ...
    (vars.amount0Signed, vars.amount1Signed, vars.sqrtPriceX96, vars.liquidity, vars.tick) =
        // @audit-issue The `sqrtPriceLimitX96` is passed directly to the KernelPool.
        IKernelPool(kernelPool).swap(
            _recipient, zeroForOne, amountSpecified, sqrtPriceLimitX96, extraData, false
        );
    // ...
}
```

This implementation differs from Uniswap V3's `SwapRouter` in two key ways:

1. **Handling of Zero Price Limit**: In Uniswap V3's `SwapRouter`, if a user provides a `sqrtPriceLimitX96` of `0`, the contract automatically substitutes it with the safest possible price limits (`TickMath.MIN_SQRT_RATIO + 1` or `TickMath.MAX_SQRT_RATIO - 1`). This allows users to perform swaps without needing to calculate a precise limit, making the interface more robust and easier to use. The Skate AMM lacks this feature, meaning a `sqrtPriceLimitX96` of `0` will cause the `KernelPool.swap(...)` call to revert;

2. **Exact Output Guarantee**: For exact output swaps, Uniswap V3's `SwapRouter` includes an on-chain check that verifies the user receives the exact requested amount of tokens if no price limit (`sqrtPriceLimitX96 == 0`) is specified. This provides an important safety guarantee against slippage. The Skate AMM's `KernelManagerLib` does not perform this check.

```
// src/UniswapV3/periphery/SwapRouter.sol
function exactOutputInternal(/*...*/) private returns (uint256 amountIn) {
    // ...
    (int256 amount0Delta, int256 amount1Delta) =
        getPool(tokenIn, tokenOut, fee).swap(
            // ...
            // @audit The router provides default price limits if none are specified.
            sqrtPriceLimitX96 == 0
                ? (zeroForOne ? TickMath.MIN_SQRT_RATIO + 1 : TickMath.MAX_SQRT_RATIO - 1)
                : sqrtPriceLimitX96,
            // ...
        );
    // ...
    // @audit-issue This check ensures the user receives the exact output amount.
    if (sqrtPriceLimitX96 == 0) require(amountOutReceived == amountOut);
}
```

By not implementing these features from the Uniswap V3 `SwapRouter`, the Skate AMM delegates the responsibility of calculating and providing a valid `sqrtPriceLimitX96` to the off-chain clients or users for every swap. It also removes the on-chain guarantee for exact output swaps when no price limit is provided.

Recommendation(s): Consider aligning the `KernelManagerLib.swap(...)` function's behavior with that of Uniswap V3's `SwapRouter`. This would involve:

1. Checking if the `sqrtPriceLimitX96` parameter is `0`. If so, replace it with the appropriate default value (`TickMath.MIN_SQRT_RATIO + 1` or `TickMath.MAX_SQRT_RATIO - 1`) based on the swap direction before calling `KernelPool.swap(...)`;
2. For exact output swaps where `sqrtPriceLimitX96` is `0`, add a requirement to ensure that the amount of tokens sent to the user matches the requested output amount;

Status: Fixed

Update from the client: The remediation is applied. commit

6.21 [Info] The `_normalizeTokenAmount(...)` function does not handle decimals larger than 18

File(s): src/amm/periphery/PeripheryPool.sol

Description: The `PeripheryPool` contract is responsible for handling user interactions on periphery chains. A core part of its logic is to normalize all token amounts to a standard `18` decimal precision before relaying them to the main `KernelManager` on the Skate chain. This is handled by the internal `_normalizeTokenAmount(...)` function.

The current implementation of this function does not support tokens with more than `18` decimals. When a token with, for example, `20` decimals is used, the calculation `NORMALIZED_DECIMAL - IERC20Metadata(token).decimals()` will attempt to compute `18 - 20`, which results in an arithmetic underflow, causing the transaction to revert.

```
function _normalizeTokenAmount(
    uint256 amount,
    address token
)
    private
    view
    returns (uint256)
{
    // @audit-issue The normalization will revert for tokens with
    // more than 18 decimals due to an underflow.
    return amount
        * 10 ** (NORMALIZED_DECIMAL - IERC20Metadata(token).decimals());
}
```

This issue affects all functions that rely on this normalization logic, such as `mint(...)`, `increaseLiquidity(...)`, and `swap(...)`, effectively preventing the use of any token with more than `18` decimals within the protocol.

Recommendation(s): Although tokens with more than `18` decimals are uncommon, if they are expected to be used within the protocol, consider adjusting the `_normalizeTokenAmount(...)` function to handle this case. Note that the `_denormalizeTokenAmount(...)` function must be adjusted as well to ensure amounts are converted back correctly.

Status: Fixed

Update from the client: The remediation is applied. commit

6.22 [Info] The `burn(...)` function lacks a check for available liquidity

File(s): solana/src/lib.rs

Description: The `burn(...)` function allows a user to initiate the burning of their liquidity position. It accepts `amount0min` and `amount1min` as parameters, which act as slippage protection by defining the minimum amount of each token the user is willing to receive. The function then constructs a `KernelAction::Burn` and dispatches it via the `action_box` program for off-chain processing.

However, the `burn(...)` function does not validate whether the pool has sufficient token balances to satisfy the `amount0min` and `amount1min` requirements before creating the action.

```
pub fn burn(
    ctx: Context<BurnCtx>,
    tick_lower: i32,
    tick_upper: i32,
    liquidity_amount: u128,
    action_box_seed: u64,
    amount0min: u64,
    amount1min: u64,
    token_id: String
) -> Result<()> {
    // @audit-issue There is no check to ensure that the pool has enough liquidity
    // to satisfy `amount0min` and `amount1min` before proceeding.

    let user_bytes32 = pubkey_to_bytes32(ctx.accounts.user.key());
    let action_id = generate_action_id(user_bytes32.clone(), ...)?;
    let kernel_action = KernelAction::Burn {
        action_id: action_id.clone(),
        chain_id: ctx.accounts.periphery_pool.chain_id,
        // ...
        amount0min,
        amount1min,
        token_id: token_id.clone(),
    };
    // ...
    // @audit An action is created even if the pool cannot fulfill the minimum amounts.
    action_box::cpi::create_action_v2(
        cpi_ctx,
        // ...
    )?;

    // ...
    Ok(())
}
```

This is inconsistent with the `decrease_liquidity(...)` function, which performs an equivalent check using the `balances_available(...)`.

```
pub fn decrease_liquidity(
    ctx: Context<DecreaseLiquidityCtx>,
    // ...
    amount0min: u64,
    amount1min: u64,
    // ...
) -> Result<()> {
    // @audit This function correctly checks for available balances.
    let (amount0_available, amount1_available) = balances_available(
        &ctx.accounts.pool_token0,
        &ctx.accounts.pool_token1,
    )?;
    require!(
        amount0min <= amount0_available && amount1min <= amount1_available,
        PeripheryError::InsufficientLiquidity
    );
    // ...
}
```

This omission allows a transaction to succeed and an action to be created, even when the pool's liquidity is insufficient to meet the user's minimum requirements. An off-chain executor will identify that the pool's balances cannot fulfill the `amount0min` and `amount1min` for settlement. Instead of executing the action in kernel and then calling `settle_burn(...)`, which would fail, the executor's likely recourse is to call `remove_staged_assets(...)` to refund the user. This results in a failed user flow where the initial `burn` transaction succeeded but did not result in a successful liquidity removal. It leads to a poor user experience, wastes the user's transaction fees, and relies on a more complex and potentially delayed recovery path.

Recommendation(s): Consider adding a balance check at the beginning of the `burn(...)` function to verify that the pool has sufficient liquidity to meet the `amount0min` and `amount1min` requirements, similar to the logic in the `decrease_liquidity(...)` function.

Status: Fixed

Update from the client: Fixed in solana commit

6.23 [Info] The `changePeripheryPool(...)` function can add new periphery pools

File(s): src/amm/kernel/libraries/KernelManagerLib.sol

Description: The `KernelManager` contract distinguishes between two roles for managing periphery pools: the `poolCreator` and the `owner`. The `poolCreator` can add a new periphery pool for a specific `kernelPool` and `chainId` by calling the `addPeripheryPool(...)` function. This function correctly ensures that a periphery pool does not already exist for the given `chainId`.

```
// src/skate/kernel/libraries/KernelManagerLib.sol
function addPeripheryPool(/*...*/)
    external
{
    // @audit Ensures that a pool does not already exist for this chainId.
    require(
        state.dataByPool[kernelPool].peripheryPoolByChainId[chainId].pool
            == bytes32(0x0),
        "periphery pool already added"
    );
    // ...
    state.dataByPool[kernelPool].peripheryPoolByChainId[chainId] = poolData;
    state.eventEmitter.peripheryPoolAdded(
        kernelPool, chainId, pool, token0, token1
    );
}
```

The `owner`, however, can call the `changePeripheryPool(...)` function. This function is intended to modify an existing periphery pool. However, it lacks a check to verify that a pool for the given `chainId` already exists. This allows the `owner` to use this function to add a new periphery pool, effectively giving it the same functionality as `addPeripheryPool(...)` but with a different access level.

```
// src/skate/kernel/libraries/KernelManagerLib.sol
function changePeripheryPool(/*...*/)
    external
{
    // @audit-issue No check to ensure a periphery pool already exists.
    DataTypes.PeripheryPool memory poolData =
        DataTypes.PeripheryPool(pool, token0, token1);

    state.dataByPool[kernelPool].peripheryPoolByChainId[chainId] = poolData;

    // @audit-issue Emits a `peripheryPoolAdded` event, not a "changed" event.
    state.eventEmitter.peripheryPoolAdded(
        kernelPool, chainId, pool, token0, token1
    );
}
```

This dual functionality is not documented and can be misleading. The function's name implies modification, not creation. Furthermore, the function emits a `peripheryPoolAdded` event, which is inaccurate when the function is used to change an existing pool's data. This ambiguity can make the codebase harder to understand and maintain.

Recommendation(s): Consider enforcing a clear separation of concerns between adding and changing periphery pools.

Status: Fixed

Update from the client: The remediation is applied. commit

6.24 [Info] The `createPool` function does not prevent creating a pool with identical tokens

File(s): `src/amm/periphery/PeripheryManager.sol`

Description: The `createPool(...)` function in the `PeripheryManager` contract is used by the `poolCreator` to deploy new periphery AMM pools. However, the function lacks a check to ensure that the two token addresses provided, `token0` and `token1`, are not the same.

This allows for the creation of a pool where both underlying assets are identical. Such a configuration serves no practical purpose in an AMM and represents an invalid state. Standard implementations, such as Uniswap V3's factory, explicitly prevent this scenario. While the `poolCreator` is a trusted role, incorporating this validation would harden the contract against accidental misconfigurations and align it with established best practices, thereby reducing the potential for errors.

Recommendation(s): Consider adding a validation check at the beginning of the `createPool(...)` function to ensure that `token0` is not equal to `token1`.

Status: Fixed

Update from the client: The remediation is applied. commit

6.25 [Info] The `settle_swap(...)` function does not utilize the `recipient` for settlement

File(s): `solana/src/lib.rs`

Description: The `swap(...)` function allows a user to specify a `recipient` for the swapped tokens, which can be different from the user initiating the transaction. Both the initiator's and the recipient's addresses are encoded and sent to the kernel chain for processing.

After the kernel processes the swap, an off-chain executor calls the `settle_swap(...)` function. The context for this function, `SettleSwapCtx`, derives the `user_data` account to be settled based on the `user` account provided in the instruction's accounts.

```
#[derive(Accounts)]
pub struct SettleSwapCtx<'info> {
    // ...
    // @audit The user_data account is derived from the `user` account,
    // which is described as the one receiving tokens in the comment below.
    #[account(
        mut,
        seeds = [b"user-data", user.key().as_ref(), periphery_pool.key().as_ref()],
        bump,
    )]
    pub user_data: Account<'info, UserDataAccount>,
    // ...
    // @audit-issue The user could be the recipient, not necessarily the swap initiator.
    /// CHECK: user who is receiving tokens / being settled
    pub user: UncheckedAccount<'info>,
}
```

As confirmed by the client during a sync call, the off-chain executor service always passes the swap initiator's address and token accounts to the `settle_swap` instruction, regardless of the `recipient` value specified in the initial `swap` call. As a result, the swap output is always transferred to the initiator, and the initiator's staged assets are the ones settled.

This effectively renders the `recipient` parameter in the `swap` function non-functional, as the specified recipient will never receive the funds. The comment in `SettleSwapCtx` is also misleading, as it suggests the `user` is the one receiving tokens, which is only true because the off-chain logic ignores the intended recipient. This behavior appears to be a workaround, as passing a recipient who has never interacted with the pool would cause the instruction to revert because their `user_data` account may not exist.

Recommendation(s): Consider either implementing the full logic to correctly handle the `recipient` address during the `settle_swap` process (which would involve passing the recipient's token accounts) or removing the `recipient` parameter from the `swap` function to avoid confusion.

Status: Fixed

Update from the client: solana commit

6.26 [Info] The `swap(...)` function does not validate the destination VM type

File(s): solana/src/lib.rs

Description: The `swap(...)` function in the `periphery_pool` contract allows a user to initiate a token swap. This action is then relayed to the Kernel on the Skate chain. The function accepts `dest_chain_id` and `dest_vm_type` to specify the target for the subsequent action.

The function correctly validates that the `dest_chain_id` matches the current chain ID. However, it fails to validate the `dest_vm_type`. An incorrect `dest_vm_type` can be provided by the user and will be passed along in the `KernelAction` sent to the Kernel chain. This could lead to processing errors for backend services that rely on the `vm_type` to correctly handle action data.

For consistency, the Solana implementation should align with the EVM implementation, which validates both the chain ID and the VM type.

The EVM `PeripheryPool` contract includes the following check:

```
// src/amm/periphery/PeripheryPool.sol
// ...
require(
    destChainId == block.chainid && destVmType == 1,
    "PeripheryPool::swap: cross-chain swap is not supported"
);
// ...
```

The Solana implementation is missing the check for `dest_vm_type`:

```
// solana/src/lib.rs
// ...
/* @audit-issue The destination VM type is not validated, unlike in the EVM implementation. */
require!(
    dest_chain_id == pool.chain_id,
    PeripheryError::InvalidChainId
);
// ...
```

Recommendation(s): Consider adding a validation check in the `swap(...)` function to ensure that the `dest_vm_type` parameter corresponds to the expected value for the Solana environment. This will improve data integrity and maintain consistency with the EVM implementation.

Status: Fixed

Update from the client: Fixed in solana commit

6.27 [Info] Tokens without a `decimals()` function are not supported

File(s): src/amm/periphery/PeripheryPool.sol

Description: The `PeripheryPool` contract includes helper functions `_normalizeTokenAmount(...)` and `_denormalizeTokenAmount(...)` to adjust token amounts to and from a normalized 18-decimal precision.

The issue is that both helper functions make a direct external call to the `decimals()` function on the token contract to get its decimal precision.

```
function _normalizeTokenAmount(uint256 amount, address token) private view returns (uint256) {
    // @audit-issue The decimals() function is optional in the ERC20 standard.
    return amount * 10 ** (NORMALIZED_DECIMAL - IERC20Metadata(token).decimals());
}

function _denormalizeTokenAmount(
    uint256 amount,
    address token
)
    private
    view
    returns (uint256)
{
    // @audit-issue The decimals() function is optional in the ERC20 standard.
    return amount / 10 ** (NORMALIZED_DECIMAL - IERC20Metadata(token).decimals());
}
```

However, the `decimals()` function is optional according to the ERC20 standard. Some valid ERC20 tokens might not implement this function. If such a token is used to create a pool, any interaction that relies on amount normalization or denormalization will revert.

Recommendation(s): Consider revisiting the logic to handle ERC20 tokens that do not implement the optional `decimals()` function. This would improve the protocol's compatibility with a wider range of tokens.

Status: Fixed

Update from the client: The remediation is applied. commit

6.28 [Info] Unsafe cast to `int256` in the `swap(...)` function

File(s): src/amm/periphery/PeripheryPool.sol

Description: The `swap(...)` function in the `PeripheryPool` contract allows users to swap tokens. As part of this process, the input amount, `amountIn`, is normalized based on the token's decimals to get `amountInNormalized`.

The problem is that this `amountInNormalized` value, which is a `uint256`, is then unsafely cast to an `int256` when constructing the `kernelAppCalldata`. If the `amountInNormalized` value exceeds the maximum value for a positive `int256` (i.e., `2^255 - 1`), the cast will not revert but will instead wrap around and be interpreted as a negative number due to two's complement representation.

This could lead to the `kernelManager` contract receiving incorrect swap data, potentially resulting in an unexpected swap outcome for the user. While the likelihood of such a large input amount is low, this behavior can become problematic, especially if combined with other vulnerabilities outlined in this report.

```
function swap(...) external override {
    // ...
    uint256 amountInNormalized = _normalizeTokenAmount(amountIn, tokenIn);
    // ...
    action.kernelAppCalldata = abi.encodeWithSignature(
        "swap(bytes)",
        abi.encode(
            // ...
            // @audit-issue The `amountInNormalized` (uint256) is unsafely cast to `int256`.
            // If it's larger than `type(int256).max`, it will become a negative value.
            int256(amountInNormalized),
            sqrtPriceLimitX96,
            extraData
        )
    );
    // ...
}
```

Recommendation(s): Consider implementing a safe casting mechanism. Before casting the `amountInNormalized` from `uint256` to `int256`, verify that its value is within the valid range for `int256`.

Status: Fixed

Update from the client: The casting issue was resolved as part of fix relating to normalization of amount specified.

6.29 [Info] `generate_action_id(...)` may return the same hash for different actions

File(s): `solana/src/lib.rs`

Description: The `periphery_pool` program includes several functions such as `mint(...)`, `increase_liquidity(...)`, `burn(...)`, `swap(...)`, and `decrease_liquidity(...)` that allow users to interact with the AMM. Each of these functions calls the internal helper function `generate_action_id(...)` to create a unique identifier for the specific action being executed.

The `generate_action_id(...)` function calculates an `action_id` by producing a Keccak256 hash from a combination of the `user`, `chain_id`, a user-provided `seed`, the `signer`, and the current blockchain `slot`.

```
pub fn generate_action_id(
    user: String,
    chain_id: u32,
    seed: u64,
    signer: Pubkey
) -> Result<String> {
    let clock: Clock = Clock::get()?;
    let slot = clock.slot;

    let mut hasher = Keccak256::new();
    hasher.update(user.as_bytes());

    hasher.update(&slot.to_be_bytes());
    let chain_id_u64 = chain_id as u64;
    hasher.update(&chain_id_u64.to_be_bytes());
    // @audit The seed is a user-controlled parameter.
    hasher.update(&seed.to_be_bytes());
    // @audit The signer is the deterministic pool authority PDA.
    hasher.update(&signer.to_bytes());

    let hash_result = hasher.finalize();
    Ok(hex::encode(hash_result))
}
```

The `signer` parameter in this context is the `pool_authority` PDA, which is deterministic for any given pool. For a specific user interacting with the same pool, the `user` and `chain_id` will also be constant. If a user's transactions are processed within the same slot, the `slot` value will be identical as well. The primary variable for ensuring uniqueness is the user-controlled `seed`.

However, if a user calls two different functions (e.g., `mint(...)` and `increase_liquidity(...)`) for the same pool within the same slot and provides the same `action_box_seed` for both calls, the `generate_action_id(...)` function will produce an identical hash. This results in two distinct user actions being assigned the same `action_id`. This could cause ambiguity and potential issues for off-chain systems that rely on unique `action_id` values for tracking and processing user actions.

Recommendation(s): Consider revisiting the logic for `action_id` generation to ensure that each distinct user action results in a unique identifier.

Status: Fixed

Update from the client: Resolved in solana commit

6.30 [Best Practices] Missing `extraData` validation in `increaseLiquidity(...)` and `mint(...)`

File(s): src/amm/periphery/PeripheryPool.sol

Description: The `mint(...)` and `increaseLiquidity(...)` functions in the `PeripheryPool` contract are used by liquidity providers to add liquidity to a pool. Both functions accept an `extraData` parameter, which is intended to contain two `uint256` values, `amount0Min` and `amount1Min`, for slippage protection.

The current implementation checks if `extraData` has a length of zero and, if so, defaults it to `abi.encode(0, 0)`. However, it does not validate that if `extraData` is provided (i.e., its length is greater than zero), it has the correct size of 64 bytes to be decoded into two `uint256` values.

```
// src/amm/periphery/PeripheryPool.sol
function mint(
    // ...
    bytes memory extraData // must contain amount0Min, amount1Min
)
    external
    override
{
    // ...
    // @audit-issue If extraData has a non-zero length but is not 64 bytes,
    // this check passes.
    if (extraData.length == 0) extraData = abi.encode(0, 0);
    // ...
    action.kernelAppCalldata = abi.encodeWithSignature(
        "mint(bytes)",
        abi.encode(
            // ...
            // @audit The potentially malformed extraData is passed to the KernelManager.
            extraData
        )
    );
    // ...
}
```

The logic for `increaseLiquidity(...)` is identical.

This malformed `extraData` is then passed to the `KernelManager` contract on the Skate chain. When the corresponding `mint(...)` or `increaseLiquidity(...)` function in `KernelManagerLib` is executed, it attempts to decode `extraData`. This `abi.decode` call will revert if the data does not have the expected length.

```
// src/amm/kernel/libraries/KernelManagerLib.sol
function _calcAmountsUsedInMintAndValidate(
    // ...
    bytes memory extraData
)
    private returns (/*...*/)
{
    // ...
    // @audit-issue This call will revert if extraData.length is not of the correct length.
    (uint256 amount0Min, uint256 amount1Min) =
        abi.decode(extraData, (uint256, uint256));
    // ...
}
```

This leads to a scenario where a user's transaction succeeds on the periphery chain, consuming gas and locking their tokens in the `PeripheryPool` contract, only to have the cross-chain operation fail later on the Skate chain. This creates a poor user experience and can cause confusion.

Recommendation(s): Consider adding a validation check in the `mint(...)` and `increaseLiquidity(...)` functions within the `PeripheryPool` contract to ensure that if `extraData` is not empty, its length is exactly 64 bytes. This would make the handling of `extraData` consistent and prevent failed transactions on the Skate chain due to malformed input.

Status: Fixed

Update from the client: The remediation is applied. commit

Update from the Nethermind Security: There are two points of concern regarding the applied fix:

- The newly added check in the `increaseLiquidity(...)` function is insufficient as it is missing the `deadline` parameter. It checks for two `uint256` values, while the `extraData` contains three;
- The sanity check that uses `abi.decode` in the expression statement might be removed by the Solidity compiler while optimizing the code;

Update from the client: Additional remediation is applied. commit

6.31 [Best Practices] The `createPool(...)` function does not validate if `tickSpacing` is configured

File(s): src/amm/kernel/libraries/KernelManagerLib.sol

Description: The `poolCreator` can create new kernel pools by calling the `createPool(...)` function in the `KernelManager` contract. This function takes a `fee` level as a parameter, which is then used to retrieve the corresponding `tickSpacing` for the new pool from the `feeAmountTickSpacing` mapping.

However, the `createPool(...)` function in the `KernelManagerLib` contract does not verify that a `tickSpacing` has been configured for the provided `fee`. If the `poolCreator` attempts to create a pool with a `fee` for which a `tickSpacing` has not been set, the lookup will default to `0`.

This will cause the deployment of the new `KernelPool` proxy to fail. The pool's `initialize(...)` function, which is called during creation, calculates `maxLiquidityPerTick` by calling `Tick.tickSpacingToMaxLiquidityPerTick(_tickSpacing)`. This function will revert with a division-by-zero error if `_tickSpacing` is `0`, leading to a failed deployment.

```
function createPool(
    DataTypes.State storage state,
    string calldata description,
    uint24 fee,
    uint160 initialPrice,
    address kernelPoolImpl
)
    external
    returns (address)
{
    // ...
    address kernelPool = address(
        new ERC1967Proxy(
            address(kernelPoolImpl),
            abi.encodeWithSignature(
                "initialize(uint160,address,address,uint24,int24)",
                initialPrice,
                token0,
                token1,
                fee,
                // @audit-issue No check to ensure tickSpacing is not 0.
                state.feeAmountTickSpacing[fee]
            )
        )
    );
    // ...
}
```

This can lead to potential operational issues if an unconfigured fee tier is used. While this does not pose a direct threat to user funds, it goes against best practices and can cause confusion.

Recommendation(s): Consider adding a validation check in the `createPool(...)` function to ensure that the `tickSpacing` for the specified `fee` is greater than zero before attempting to create the pool. This aligns with the implementation in Uniswap V3's factory contract and prevents the creation of broken pools.

Status: Fixed

Update from the client: The remediation is applied. commit

6.32 [Best Practices] The `setFeeAmountTickSpacing(...)` function lacks input validation

File(s): src/amm/kernel/libraries/KernelManagerLib.sol

Description: The `owner` of the `KernelManager` contract can configure new fee tiers by calling the `setFeeAmountTickSpacing(...)` function. This function establishes a link between a swap `fee` and its corresponding `tickSpacing`.

The current implementation lacks any validation on the `fee` and `spacing` inputs. This allows the `owner` to set a `tickSpacing` of `0` or an excessively large value, which can be configured for any given `fee`.

```
function setFeeAmountTickSpacing(
    DataTypes.State storage state,
    uint24 fee,
    int24 spacing
)
    external
{
    // @audit-issue No validation on fee or spacing parameters.
    state.feeAmountTickSpacing[fee] = spacing;
}
```

The `KernelPool` contract relies on Uniswap V3's math libraries for its core operations. These libraries, particularly those related to tick bitmaps and calculations, operate under the assumption that `tickSpacing` is a positive, reasonably-sized number. For instance, Uniswap V3's implementation caps the `tickSpacing` to prevent potential overflows in functions like `TickBitmap.nextInitializedTickWithinOneWord`. Allowing arbitrary values could lead to unexpected reverts or incorrect calculations during swaps and liquidity management operations.

Recommendation(s): Consider introducing input validation for the `fee` and `spacing` parameters in the `setFeeAmountTickSpacing(...)` function. It is recommended to constrain the `tickSpacing` to a safe, positive range that is compatible with the underlying Uniswap V3 math libraries used by the `KernelPool`, for example by ensuring `spacing` is greater than `0` and less than a reasonable upper bound.

Status: Fixed

Update from the client: The remediation is applied. commit

6.33 [Best Practices] Usage of `tx.origin` for authorization is unsafe

File(s): src/skate/kernel/MessageBox.sol

Description: The `submitTasks(...)` function in the `MessageBox` contract is responsible for processing user intents submitted by off-chain executors. Authorization is performed by checking if `tx.origin` is a whitelisted executor.

Relying on `tx.origin` for authorization can make contracts vulnerable to phishing attacks. If a whitelisted executor is lured into calling a malicious contract, that contract could then call `submitTasks(...)` on the executor's behalf. Since `tx.origin` would still be the executor's address, the malicious call would pass the authorization check, potentially leading to the submission of unauthorized tasks.

```
function submitTasks(
    IMessageBox.Task[] calldata tasks,
    IMessageBox.Intent calldata intent
)
    external
    override
{
    // @audit-issue The check relies on tx.origin, which can be manipulated in phishing attacks.
    require(_executorRegistry.isExecutor(tx.origin), NotAnExecutor(tx.origin));
    require(msg.sender == intent.intentData.appAddress, IntentIsNotSignedForTheApp());
    // ...
}
```

Recommendation(s): Consider replacing the `tx.origin` check with a `msg.sender` check. Since `submitTasks(...)` is intended to be called by `SkateApp` contracts, the authorization logic could be moved to the `processIntent(...)` function within the `SkateApp` contract, verifying that `msg.sender` is a whitelisted executor.

Status: Acknowledged

Update from the client: We won't be making changes to the current architecture in this version. For the current version, we are running the executors and the executor interaction would be only be through a set of applications. Additionally, the off-chain process also validates the authenticity of applications emitting the tasks. The design-wise, we did not want to give this authorization control to the applications since they can make modification to avoid it e.g. if this check is moved to "processIntent" function, the application logic can override the function and circumvent this authorization check.

7 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 the development and maintenance of smart contracts. They help ensure that the contract is properly designed, implemented, and tested and provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Skate AMM's documentation

The Skate team provided a Notion document outlining a high-level overview of the inner workings and flow of the contracts. The team was also available during synchronous calls to answer any questions the Nethermind Security team had.

However, given the overall complexity of the project, the provided documentation lacks the necessary depth. The contracts heavily rely on external backend logic that is currently undocumented. This makes it challenging to validate the correctness of function input arguments without knowledge of the backend's implementation details and assumptions.

It is strongly advised to improve the documentation, especially in the areas where there are assumptions about the backend operating in a particular way. This will not only help external readers understand the system but also facilitate better coordination between smart contract and backend engineers within the Skate team, reducing the potential for mistakes in future code iterations. A good starting point would be to expand the NatSpec comments within the code.

8 Test Suite Evaluation

Remarks about Skate AMM's test suite

The provided test suite for the Skate AMM project lacks sufficient depth and coverage, with several areas needing improvement.

For the EVM components, it was noted that certain core flows in the protocol are untested. Implementing comprehensive end-to-end tests that cover primary user flows is essential for a project of this complexity. Furthermore, to ensure the robustness of the AMM logic, it would be highly beneficial to adapt Uniswap's comprehensive test suite to thoroughly validate the functionality of the `KernelPool` contract.

Regarding the Solana program, the provided code was taken out of a typical Solana project folder structure. As a result, the Nethermind Security team was unable to compile and test the project in the provided form.

8.1 Compilation Output

```
> forge build
[] Compiling...
[] Compiling 145 files with Solc 0.8.26
[] Solc 0.8.26 finished in 127.71s
Compiler run successful with warnings:
Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> script/common/test.s.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> script/common/view.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> script/periphery/burn.s.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> script/periphery/mint.s.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> script/periphery/tETH/mint.s.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> test/amm/TestUpgradeability.t.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> test/amm/testView.t.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.26;"
--> test/skate/kernel/MessageBox.t.sol

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/Integration.t.sol:486:10:
    |
486 |         (uint256 amount0, uint256 amount1,) = manager.lensBurn(1, liquidity);
    |          ^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/Integration.t.sol:48:5:
   |
48 |     uint256 amount0 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/Integration.t.sol:486:27:
    |
486 |         (uint256 amount0, uint256 amount1,) = manager.lensBurn(1, liquidity);
    |                           ^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/Integration.t.sol:49:5:
   |
49 |     uint256 amount1 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/Integration.t.sol:607:10:
    |
607 |         (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = manager.lensSwap(
    |          ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/Integration.t.sol:48:5:
   |
48 |     uint256 amount0 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/Integration.t.sol:607:26:
    |
607 |         (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = manager.lensSwap(
    |                          ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/Integration.t.sol:49:5:
   |
49 |     uint256 amount1 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:519:14:
    |
519 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:34:5:
   |
34 |     uint256 amount0 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:519:30:
    |
519 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |                              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:35:5:
   |
35 |     uint256 amount1 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:605:14:
    |
605 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:34:5:
   |
34 |     uint256 amount0 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:605:30:
    |
605 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |                              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:35:5:
   |
35 |     uint256 amount1 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:667:14:
```

```
667 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:34:5:
   |
34 |     uint256 amount0 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:667:30:
    |
667 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |                              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:35:5:
   |
35 |     uint256 amount1 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:729:14:
    |
729 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:34:5:
   |
34 |     uint256 amount0 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/kernel/kernel-manager.t.sol:729:30:
    |
729 |             (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = kernelManager.lensSwap(
    |                              ^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/kernel/kernel-manager.t.sol:35:5:
   |
35 |     uint256 amount1 = 100 ether;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/periphery/periphery-pool.t.sol:176:9:
    |
176 |         uint256 amount0 = 10 * 10 ** 18;
    |         ^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/periphery/periphery-pool.t.sol:35:5:
   |
35 |     uint256 amount0 = 10 * 10 ** 6;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/amm/unit/periphery/periphery-pool.t.sol:177:9:
    |
177 |         uint256 amount1 = 10 * 10 ** 18;
    |         ^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/amm/unit/periphery/periphery-pool.t.sol:36:5:
   |
36 |     uint256 amount1 = 10 * 10 ** 8;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> src/amm/kernel/KernelPool.sol:434:9:
    |
434 |         bytes calldata data,
    |         ^^^^^^^^^^^^^^^^^^^

Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> src/amm/kernel/KernelPool.sol:586:9:
```

```
586 |         address recipient,
    |         ^^^^^^^^^^^^^^^^^

Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> src/amm/kernel/KernelPool.sol:590:9:
    |
590 |         bytes calldata data,
    |         ^^^^^^^^^^^^^^^^^^^

Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> src/amm/periphery/PeripheryPool.sol:81:32:
   |
81 |     function _authorizeUpgrade(address newImplementation) internal override(UUPSUpgradeable) {
   |                                 ^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/kernel/add-periphery-pools.sol:20:9:
   |
20 |         address kernelPool = 0x25c2478a7d42A25CAbdC3B763bD7C6a71cDE3424;
   |         ^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/kernel/deploy-kernel-pool.s.sol:15:9:
   |
15 |         address executor = 0x7A362dEaE4e75D6C7FcAf8B4505a1A846b3BeE1d;
   |         ^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/swap.s.sol:18:9:
   |
18 |         address user = 0x7A362dEaE4e75D6C7FcAf8B4505a1A846b3BeE1d;
   |         ^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/swap.s.sol:19:9:
   |
19 |         address kernelManager = 0x31697549Cab09fD99CB2D9115fA92c3411D0Db1C;
   |         ^^^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/swap.s.sol:20:9:
   |
20 |         address kernelPool = 0xB67C497787711E32517449F8b5711031eA3D4128;
   |         ^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/swap.s.sol:21:9:
   |
21 |         PeripheryPool pool = PeripheryPool(0xBfF5a7e33AdE7367Da35c228bA7FD573a13c61Bf);
   |         ^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/swap.s.sol:22:9:
   |
22 |         ISkateGateway gateway = ISkateGateway(0x3823143157F16303BBde521Ff1b7a2b2179E9d44);
   |         ^^^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/swap.s.sol:24:9:
   |
24 |         uint256 peripheryForkId = vm.createFork(vm.envString("RPC_MAINNET"));
   |         ^^^^^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/swap.s.sol:25:9:
   |
25 |         uint256 skateForkId = vm.createFork("https://rpc.skatechain.org");
   |         ^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/usdc-usdt/deployment.s.sol:20:9:
```

```
20 |         address executor = 0x7A362dEaE4e75D6C7FcAf8B4505a1A846b3BeE1d;
   |         ^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> script/periphery/usdc-usdt/deployment.s.sol:26:9:
   |
26 |         address kernelManager = 0x31697549Cab09fD99CB2D9115fA92c3411D0Db1C;
   |         ^^^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
   --> test/amm/Integration.t.sol:607:10:
    |
607 |         (int256 amount0, int256 amount1, uint160 sqrtPriceX96After,) = manager.lensSwap(
    |          ^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
   --> test/amm/unit/periphery/periphery-pool.t.sol:318:9:
    |
318 |         uint128 liquidityAmount = 1;
    |         ^^^^^^^^^^^^^^^^^^^^^^^

Warning (2018): Function state mutability can be restricted to view
  --> src/amm/periphery/PeripheryPool.sol:81:5:
   |
81 |     function _authorizeUpgrade(address newImplementation) internal override(UUPSUpgradeable) {
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/amm/unit/kernel/kernel-manager.t.sol:91:5:
   |
91 |     function testKernelManager_Deployment() public {
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/amm/unit/kernel/kernel-pool.t.sol:46:5:
   |
46 |     function testKernelPool_Deployment() public {
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/amm/unit/periphery/periphery-manager.t.sol:36:5:
   |
36 |     function testPeripheryManager_KernelManagerDeployment() public {
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/amm/unit/periphery/periphery-pool.t.sol:75:5:
   |
75 |     function testPeripheryPool_Deployment() public {
   |     ^ (Relevant source part starts here and spans across multiple lines).
```

8.2 Tests Output

```
> forge test
Ran 8 tests for test/amm/unit/periphery/periphery-manager.t.sol:PeripheryManagerTest
[PASS] testPeripheryManager_KernelManagerDeployment() (gas: 45005)
[PASS] testPeripheryManager_OnlyOwnerCanUpgrade() (gas: 1625007)
[PASS] testPeripheryManager_OwnerCanCreatePeripheryPool() (gas: 3143166)
[PASS] testPeripheryManager_OwnerCanEnableNewFeeTier() (gas: 49382)
[PASS] testPeripheryManager_OwnerCanSetNewActionBox() (gas: 31409)
[PASS] testPeripheryManager_OwnerCanSetNewEventEmitter() (gas: 31769)
[PASS] testPeripheryManager_OwnerCanUpgradePeripheryPool() (gas: 5902211)
[PASS] testPeripheryManager_itialize() (gas: 16469)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 14.55ms (8.31ms CPU time)

Ran 15 tests for test/amm/unit/kernel/kernel-manager.t.sol:KernelManagerTest
[PASS] testKernelManager_Burn() (gas: 1163184)
[PASS] testKernelManager_CollectProtocolFee() (gas: 1674471)
[PASS] testKernelManager_DecreaseLiquidity() (gas: 1195614)
[PASS] testKernelManager_Deployment() (gas: 18995)
[PASS] testKernelManager_IncreaseLiquidity() (gas: 1360870)
[PASS] testKernelManager_Mint() (gas: 1158771)
[PASS] testKernelManager_OnlyOwnerCanCreatePool() (gas: 17953)
[PASS] testKernelManager_OnlyOwnerCanRegisterPeripheryPool() (gas: 114461)
[PASS] testKernelManager_OnlyOwnerCanSetEventEmitter() (gas: 35689)
[PASS] testKernelManager_OnlyOwnerCanSetFeeTiers() (gas: 53534)
[PASS] testKernelManager_OnlyOwnerCanSetProtocolFee() (gas: 68412)
[PASS] testKernelManager_OnlyOwnerCanUpgrade() (gas: 4325762)
[PASS] testKernelManager_OnlyOwnerCanUpgradePool() (gas: 4912110)
[PASS] testKernelManager_Swap() (gas: 1596413)
[PASS] testKernelManager_itialize() (gas: 16357)
Suite result: ok. 15 passed; 0 failed; 0 skipped; finished in 20.65ms (19.58ms CPU time)

Ran 10 tests for test/amm/unit/periphery/periphery-pool.t.sol:PeripheryPoolTest
[PASS] testPeripheryPool_Burn() (gas: 1317215)
[PASS] testPeripheryPool_BurnAndSettle() (gas: 1301710)
[PASS] testPeripheryPool_Deployment() (gas: 45096)
[PASS] testPeripheryPool_MintAndSettle() (gas: 788106)
[PASS] testPeripheryPool_MintWithEighteenDecimals() (gas: 4761825)
[PASS] testPeripheryPool_MintWithNonEighteenDecimals() (gas: 832469)
[PASS] testPeripheryPool_Swap() (gas: 3530477)
[PASS] testPeripheryPool_SwapAndSettle() (gas: 1441517)
[PASS] testPeripheryPool_changeKernelAddresses() (gas: 61603)
[PASS] testPeripheryPool_itialize() (gas: 28197)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 25.88ms (14.42ms CPU time)

Ran 6 tests for test/amm/Integration.t.sol:IntegrationTest
[PASS] testIntegration_Burn() (gas: 3716057)
[PASS] testIntegration_CollectProtocol() (gas: 2971356)
[PASS] testIntegration_DecreaseLiquidity() (gas: 3331139)
[PASS] testIntegration_IncreaseLiquidity() (gas: 2776054)
[PASS] testIntegration_Mint() (gas: 1871283)
[PASS] testIntegration_Swap() (gas: 2833006)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 25.94ms (44.14ms CPU time)

Ran 8 tests for test/amm/unit/kernel/kernel-pool.t.sol:KernelPoolTest
[PASS] testKernelPool_Deployment() (gas: 28412)
[PASS] testKernelPool_OnlyManagerCanBurn() (gas: 283404)
[PASS] testKernelPool_OnlyManagerCanCollectProtocol() (gas: 38229)
[PASS] testKernelPool_OnlyManagerCanMint() (gas: 303818)
[PASS] testKernelPool_OnlyManagerCanSetFeeProtocol() (gas: 39033)
[PASS] testKernelPool_OnlyManagerCanSwap() (gas: 3127852)
[PASS] testKernelPool_OnlyOwnerCanUpgrade() (gas: 4894626)
[PASS] testKernelPool_itialize() (gas: 20764)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 31.96ms (22.37ms CPU time)

Ran 5 test suites in 112.16ms (118.98ms CPU time): 47 tests passed, 0 failed, 0 skipped (47 total tests)
```

9 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 conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to 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.

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.