Ekubo EVM Protocol v2 Audit Report

Prepared by Riley Holterhus

November 18, 2025

Contents

1 Introduction

1.1 About Ekubo

Ekubo is a high-performance, gas-optimized AMM. It supports several configuration options for pools, including choices between concentrated and stableswap liquidity, hookable extensions, and parameters such as fees and tick spacing. There is a Starknet version of the protocol and an EVM version currently deployed on Ethereum mainnet. An upcoming set of v2 EVM contracts will introduce further gas optimizations and new features, and is intended to be deployable on additional EVM chains.

1.2 About the Auditor

Riley Holterhus is an independent security researcher who focuses on Solidity smart contracts. Other than conducting independent security reviews, he works as a Lead Security Researcher at Spearbit, and also searches for vulnerabilities in live codebases. Riley can be reached by email at rileyholterhus@gmail.com, by Telegram at @holterhus and on Twitter/X at @rileyholterhus.

1.3 Disclaimer

2 Audit Overview

2.1 Scope of Work

From October 13 to November 12, 2025, Riley Holterhus conducted a manual security review of Ekubo's `evm-contracts`. The goal was to identify potential vulnerabilities and logic issues in the protocol.

The audit was performed on the codebase found in the `EkuboProtocol/evm-contracts` GitHub repository (private at the time of writing). The audit started on commit dca6d24, and all files were considered in scope. Particular attention was given to the v2 changes to the protocol, along with components not previously reviewed by the auditor, including the `MEVCapture` and `TWAMM` extensions. The last in-scope commit reviewed was commit 2c32cb9.

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "Critical", "High", "Medium", "Low" or "Informational". These severities aim to capture the impact and likelihood of each potential issue. A separate section is included for "Gas Optimizations". An appendix of known issues and known behavior is also provided at the end of the report.

In total, **15 findings** were identified: 2 Critical, 1 High, 4 Medium, 3 Low, 2 Informational, and 3 related to Gas Optimizations.

3 Findings

3.1 Critical Severity Findings

3.1.1 `startPayments()` can be tricked into storing zero starting balance

Description: The `startPayments()` and `completePayments()` functions record the contract's `balanceOf()` tokens at two points. `startPayments()` stores the current balance in transient storage, and `completePayments()` later checks the balance again and credits the caller with the difference if positive. The implementation of the `balanceOf()` call is:

```
let tokenBalance :=
    mul( // The arguments of `mul` are evaluated from right to left.
        mload(returnLocation),
        and( // The arguments of `and` are evaluated from right to left.
            gt(returndatasize(), 0x1f), // At least 32 bytes returned.
            staticcall(gas(), token, 0x10, 0x24, returnLocation, 0x20)
        )
    )
```

Notice that this sets `tokenBalance` to zero if the `balanceOf()` call fails.

An attacker could exploit this by selecting a token that Ekubo already holds and deliberately causing the `balanceOf()` call to fail during `startPayments()`. This would result in a starting balance of zero being recorded in transient storage. If `balanceOf()` then succeeds in `completePayments()` and returns the actual balance, the attacker would be credited with that amount despite not transferring anything.

One possible way to get the `balanceOf()` call to fail in `startPayments()` is to deliberately limit the gas provided to the call. Due to EIP-150, the `balanceOf()` staticcall can only forward at most 63/64 of the remaining gas. If the token's `balanceOf()` is expensive enough where a 63/64 amount is insufficient while the leftover 1/64 is enough gas for the rest of `startPayments()`, the inner call will revert but the entire function will not. This was demonstrated in a PoC submitted to Ekubo using the aUSDC token, which has a relatively expensive `balanceOf()` implementation.

Note that this behavior is much less likely to be an issue in the previous version of Ekubo, since it performed the equivalent of `startPayments()` and `completePayments()` within a single call. This meant the gas limits for the calls couldn't be chosen independently.

Recommendation: Consider changing the logic to not allow a failed `balanceOf()` call in `startPayments()` to be equivalent to a zero balance.

Ekubo: Fixed in PR 299.

Auditor: Verified. The change was made specifically to `startPayments()`, and `completePayments()` is kept as is. Now if `startPayments()` observes a failure in the `balanceOf()` call, it becomes a no-op and does not record a zero balance that can later be used in a `completePayments()` call.

3.1.2 `setExtraData()` allows arbitrary storage writes

Description: The `setExtraData()` function lets users write to their `extraData` field (a `bytes16`) in their position structs. This function works by calculating the position struct's storage slot and replacing its lower 16 bytes with the provided value:

```
// In Core.sol
function setExtraData(PoolId poolId, PositionId positionId, bytes16 _extraData) external {
    StorageSlot firstSlot = CoreStorageLayout.poolPositionsSlot(poolId, msg.sender, positionId);

    bytes32 extraData;
    assembly ("memory-safe") {
        extraData := _extraData
    }

    firstSlot.store(((firstSlot.load() >> 128) << 128) | (extraData >> 128));
}

// In CoreStorageLayout.sol
function poolPositionsSlot(PoolId poolId, address owner, PositionId positionId)
    internal
    pure
    returns (StorageSlot firstSlot)
{
    assembly ("memory-safe") {
        mstore(0, positionId)
        firstSlot := add(keccak256(0, 32), add(poolId, owner))
    }
}
```

In this code, both `poolId` and `positionId` are arbitrary `bytes32` values with no additional validation. This means `poolPositionsSlot()` can reach any slot of the form `add(keccak256(positionId), add(poolId, owner))`. For any target slot `s`, an attacker can simply set `poolId := sub(sub(s, keccak256(positionId)), owner)` so that `setExtraData()` writes to slot `s`.

This allows arbitrary storage writes, so an attacker could use this to drain the contract by writing to critical state, such as internal token balances.

Recommendation: One mitigation is to accept a `PoolKey` instead of a `PoolId` as input, then call `poolKey.validate()` followed by `poolKey.toPoolId()` to get the `poolId`, and finally `positionId.validate(poolKey.config)`. This enforces that both ids are properly formed and ensures the `poolId` is derived from a hash as expected.

Another option is to change the way `poolPositionsSlot()` derives the storage slot, for example by adding another layer of hashing, so it can't overlap with any other storage slot calculation.

Ekubo: Addressed by changing the `poolPositionsSlot()` calculation in commit 3695bc3 and commit 5e41a82.

Auditor: Verified. The `poolPositionsSlot()` slot is now derived as `keccak256(keccak256(positionId ++ poolId ++ owner) ++ bytes32(1))`. The additional hash using `bytes32(1)` ensures that the first 96-byte hash cannot overlap with the `poolStateSlot()`, which is also derived from a 96-byte hash.

3.2 High Severity Findings

3.2.1 `MEVCapture` can be repeatedly initialized

Description: The `beforeInitializePool()` function in the MEVCapture contract is missing the `onlyCore` modifier, which means it can be called by anyone after the pool has already been initialized. This allows an attacker to repeatedly call the function to reset `_lastUpdateTime` to the current `block.timestamp`, and set `_tickLast` to any value they choose.

This behavior breaks the internal accounting. One way an attacker could abuse this is by setting `_tickLast` to a highly out-of-range value, forcing honest users to pay significantly higher fees. Ideally, slippage checks would catch this attack. Another potential abuse is repeatedly updating `_lastUpdateTime` to let fees accumulate over time, then capturing all of those fees by swapping into an extreme tick range where the attacker is the only active LP.

Recommendation: Add the `onlyCore` modifier to the `beforeInitializePool()` function.

Ekubo: Fixed in commit a4811d4.

Auditor: Verified.

3.3 Medium Severity Findings

3.3.1 `updateDebt()` can be misused by nonstandard tokens

Description: The `updateDebt()` function is intended to be called by token contracts, with the `msg.sender` being treated as the token. Since this function can freely modify the debt associated with the token, the design is assuming that any token able to call `updateDebt()` was explicitly designed to integrate with the system and will behave correctly.

This assumption can be risky, and some tokens with large market caps have unusual behaviors that could interfere with this system. One notable example is SAND, a token with a $500M market cap at the time of writing. SAND allows users to make arbitrary external calls from the token contract, so long as the first parameter of the call equals `msg.sender`:

```
function approveAndCall(
    address target,
    uint256 amount,
    bytes calldata data
) external payable returns (bytes memory) {
    require(
        BytesUtil.doFirstParamEqualsAddress(data, msg.sender),
        "first param != sender"
    );

    _approveFor(msg.sender, target, amount);

    // solium-disable-next-line security/no-call-value
    (bool success, bytes memory returnData) = target.call.value(msg.value)(data);
```

```
        require(success, string(returnData));
        return returnData;
    }

    function doFirstParamEqualsAddress(bytes memory data, address _address)
        internal
        pure
        returns (bool)
    {
        if (data.length < (36 + 32)) {
            return false;
        }
        uint256 value;
        assembly {
            value := mload(add(data, 36))
        }
        return value == uint256(_address);
    }
```

This behavior means an attacker could instruct SAND to call `updateDebt()` and manipulate the debt associated with SAND in Ekubo. However, note that technically the attacker's address would need to be the delta in `updateDebt()`, which means the delta would be positive, which is not as useful for an exploit.

That said, the SAND example shows that the current design makes assumptions about token behavior that may not hold.

Recommendation: Following a discussion with Ekubo, it was decided that the `updateDebt()` function could enforce a non-standard calldata encoding to reduce the risk of unintended usage.

Ekubo: Fixed as described above in PR 294.

Auditor: Verified. The approach taken is to require that the calldata length is exactly 20 bytes: 4 bytes for the selector and 16 bytes for the delta. This change mitigates the issue demonstrated with SAND and lowers the likelihood that other non-standard tokens could interact with `updateDebt()` in unexpected ways.

3.3.2 `addConstrainSaleRateDelta()` logic can create unwithdrawable orders

Description: When a TWAMM order is created or updated, its net impact on the global sale rate is queued by adjusting the `saleRateDelta` at its start and end timestamps. Specifically, the order adds a positive delta at the start timestamp (where its sale rate begins contributing) and a negative delta at the end timestamp (where it stops contributing).

Each time a `saleRateDelta` is updated at an initialized timestamp, the `_addConstrainSaleRateDelta()` helper function is called:

```
    function _addConstrainSaleRateDelta(int112 saleRateDelta, int256 saleRateDeltaChange)
        internal
        pure
        returns (int112 saleRateDeltaNext)
```

```
{
    int256 result = int256(saleRateDelta) + saleRateDeltaChange;

    // checked addition, no overflow of int112 type
    if (FixedPointMathLib.abs(result) > MAX_ABS_VALUE_SALE_RATE_DELTA) {
        revert MaxSaleRateDeltaPerTime();
    }

    // we know cast is safe because abs(result) is less than MAX_ABS_VALUE_SALE_RATE_DELTA which
        fits in a int112
    saleRateDeltaNext = int112(result);
}
```

This function ensures that the net `saleRateDelta` at any timestamp does not exceed `MAX_ABS_VALUE_SALE_RATE_DELTA` in absolute value. This cap is important to prevent the global sale rate from overflowing.

Since this constraint is only applied to the *net* delta, not the *gross* contributions that make it up (which isn't currently tracked), it's possible for a position to become unwithdrawable if its contribution is the only thing keeping the net value within bounds. For example, suppose the following orders are created:

- User 1 creates an order over time interval `[a, b]`
- User 2 creates an order over time interval `[b, c]`
- User 3 creates an order over time interval `[b, d]`

If user 1 tries to cancel their order, it could revert, since their negative delta at timestamp `b` might be needed to keep the net rate within bounds given the positive contributions from users 2 and 3.

Recommendation: One potential fix is to track the gross sale rate at each timestamp and apply the constraint to that instead of the net value. This would require additional logic and likely more storage to maintain the gross totals separately.

Ekubo: Acknowledged that this is a real issue, but we decided not to store the additional state required to prevent it due to the increased gas cost, the large amounts of tokens involved at the maximum sale rate and the medium impact when the issue occurs. The maximum sale rate of a token with 18 decimals is approximately 13,284 tokens per second. For tokens like ETH and EKUBO, these sale rates are not practically executed on TWAMM pools which only support full range liquidity. For tokens with decimals < 18, it's even less likely to come up.

An attacker can intentionally grief an existing order, but in practice they would have to lock up more capital due to the minimum order duration for an order that starts in the future, and also have to pay the pool fee to avoid executing their own order.

This could practically occur with low value tokens that have large total supplies and/or decimals > 18, but the impact is lower in these cases since the value of the tokens being sold at the maximum sale rate is smaller, all things equal.

Auditor: Acknowledged. Also note that the above note from Ekubo is after the minimum order duration was increased in PR 300, so griefers now need to lock up their capital for longer than previously was required.

3.3.3 TWAMM order state can collide with pool state

Description: The TWAMM contract stores order state in the storage slot `keccak256(owner ++ salt ++ orderId)`, which is a 96-byte hash input. It also stores pool state at the storage slot equal to `poolId`, and each `poolId` is itself computed as `keccak256(poolKey)`, which is another 96-byte hash. Because both slots come from 96-byte hash inputs, there is a chance of collision.

```
// In TWAMMStorageLayout.sol
function orderStateSlotFollowedByOrderRewardRateSnapshotSlot(address owner, bytes32 salt, OrderId
    orderId)
    internal
    pure
    returns (StorageSlot slot)
{
    assembly ("memory-safe") {
        let free := mload(0x40)
        mstore(free, owner)
        mstore(add(free, 0x20), salt)
        mstore(add(free, 0x40), orderId)
        slot := keccak256(free, 96)
    }
}

// In TWAMMStorageLayout.sol
function twammPoolStateSlot(PoolId poolId) internal pure returns (StorageSlot slot) {
    slot = StorageSlot.wrap(PoolId.unwrap(poolId));
}

// In poolKey.sol
function toPoolId(PoolKey memory key) pure returns (PoolId result) {
    assembly ("memory-safe") {
        // it's already copied into memory
        result := keccak256(key, 96)
    }
}
```

There is at least one way this can be exploited or is close to exploitable. The TWAMM function `lockAndExecuteVirtualOrders()` accepts an arbitrary `poolKey` and eventually calls `_executeVirtualOrdersFromWithinLock()` with the calculated `poolId`. An attacker can collide with a given order's state by choosing a poolKey such that:

- `poolKey.token0` is equal to the order owner
- `poolKey.token1` is equal to the order salt
- `poolKey.config` is equal to the order id

This would hash the same way the order was hashed when it was written to storage.

Within the time available during the audit there was no critical exploit proven from here. There are some hurdles to building a full exploit, for example the chosen `poolKey` is not a real pool, so no swaps can be executed in `lockAndExecuteVirtualOrders()`. However a PoC was provided that showed an order's `lastUpdateTime` could be overwritten via this behavior.

Recommendation: Consider changing the hashing calculations to ensure that a collision cannot happen even with arbitrary `poolKey` inputs.

Ekubo: Fixed in commit e4cc13f.

Auditor: Verified. The order state now applies an offset to the storage slot after hashing, preventing collisions from occurring in this way.

3.3.4 `saleRateDelta` casting can overflow

Description: In the `Orders` contract, the `handleLockData()` function decodes `saleRateDelta` as an `int256` but later casts it to `int112` before passing it to `updateSaleRate()`.

```
function handleLockData(/* ... */) /* ... */ {
    // ...
    if (callType == CALL_TYPE_CHANGE_SALE_RATE) {
        (, /* ... */, /* ... */, /* ... */, int256 saleRateDelta) =
            abi.decode(data, (uint256, address, uint256, OrderKey, int256));
        int256 amount = CORE.updateSaleRate(/* ... */, /* ... */, /* ... */, int112(saleRateDelta))
            ;
        // ...
    }
    // ...
}
```

The two functions that can reach this logic are `increaseSellAmount()` and `decreaseSaleRate()`. In both cases, it's possible for the `saleRateDelta` to exceed the valid range for an `int112`. For example, in `decreaseSaleRate()`, a user can provide input such that `saleRateDelta` becomes `-type(uint112).max`, which would overflow when cast to `int112`. This could cause the logic to do the opposite of what was intended, such as increasing the sale rate instead of decreasing it.

The most dangerous outcome would be if this caused tokens to move in the wrong direction. For example, if a "decrease" in sale rate was treated as an increase and funds were taken from the `recipientOrPayer`. However, this can't happen because the decision between `payFrom()` and `withdraw()` is based on the original `int256 saleRateDelta`, not the truncated `int112` value.

So, the impact appears to be limited to cases where users provide extremely large values and unintentionally trigger the opposite action.

Recommendation: To ensure that this behavior can't be exploited, consider changing the `int112` cast into a safe cast, so it explicitly reverts on overflows.

Ekubo: Fixed in commit d0cabe3.

Auditor: Verified. The code now has a safe cast.

3.4 Low Severity Findings

3.4.1 `BasePositions` can reach `int128` overflow

Description: In the `BasePositions` contract, the `CALL_TYPE_WITHDRAW` case allows users to supply an arbitrary `uint128 liquidity` value. This value is later cast to `int128` and negated in the call to `updatePosition()`:

```
PoolBalanceUpdate balanceUpdate = CORE.updatePosition(
    poolKey,
    createPositionId({_salt: bytes24(uint192(id)), _tickLower: tickLower, _tickUpper: tickUpper}),
    -int128(liquidity)
);
```

If the user provides a value larger than `type(int128).max`, the cast will overflow and wrap into a negative value. The negation then turns it back into a positive number, causing `updatePosition()` to add liquidity instead of removing it. This also flips the expected sign of the return values, which may cause further casting issues in downstream logic.

This will likely end in a revert due to unpaid debt from the liquidity addition. However, there is some risk due to the later call to `updateSavedBalances()`, which is intended to save newly accrued fees. If this path can reach a casting issue as well, it might instead load already-collected fees, which would be stealable by the caller.

Recommendation: Consider explicitly checking that `liquidity` is not larger than `type(int128).max`.

Ekubo: Fixed in commit 973b99d.

Auditor: Verified.

3.4.2 `toDropState()` doesn't mask value before packing

Description: The `toDropState()` function packs two `uint128` values:

```
function toDropState(uint128 _funded, uint128 _claimed) pure returns (DropState state) {
    assembly ("memory-safe") {
        state := or(shl(128, _funded), _claimed)
    }
}
```

The `_claimed` value is not masked, which could allow dirty upper bits to affect the result. While this isn't an issue since the function is unused, most other packing functions in the codebase do apply masking.

Recommendation: If this function is used in the future with inputs where `_claimed` might contain dirty upper bits, consider masking out the top 128 bits. For example:

```
state := or(shl(128, _funded), shr(128, shl(128, _claimed)))
```

Ekubo: Addressed by removing the `toDropState()` function in commit eef6437.

Auditor: Verified.

3.4.3 MEVCapture reverts on `isFullRange()` instead of `isStableswap()`

Description: The `beforeInitializePool()` function in the MEVCapture contract currently reverts if the given `poolKey` corresponds to a full-range pool. However, this check is too narrow. The intended behavior is to revert for any pool that is not a concentrated liquidity pool. Full-range pools are just one specific type of non-concentrated pool with an amplification factor and a center of zero. There are other non-concentrated pool types that would incorrectly pass this check.

Recommendation: Change the logic to revert if `isStableswap()` is true instead:

```diff
- if (poolKey.config.isFullRange()) {
+ if (poolKey.config.isStableswap()) {
      revert ConcentratedLiquidityPoolsOnly();
  }
```

This is the more generic check that was intended.

Ekubo: Fixed in commit a4811d4.

Auditor: Verified.

3.5 Informational Findings

3.5.1 Incorrect comments in `RevenueBuybacksLib`

Description: The comment above `RevenueBuybacksLib` incorrectly refers to it as an “Oracle Library”, likely these are leftover comments from adapting the existing `OracleLib`.

Recommendation: Update the comments above `RevenueBuybacksLib` and remove the mentions of `OracleLib`.

Ekubo: Fixed in commit 038cb39.

Auditor: Verified.

3.5.2 Incorrect comments for `MEVCapturePoolState` bits

Description: The `createMEVCapturePoolState()` function is as follows:

```solidity
function createMEVCapturePoolState(uint32 _lastUpdateTime, int32 _tickLast) pure returns (
    MEVCapturePoolState s) {
    assembly ("memory-safe") {
        // s = (lastUpdateTime << 224) | (tickLast << 192)
```

```
            s := or(shl(224, _lastUpdateTime), and(_tickLast, 0xffffffff))
        }
    }
```

The comment above the main assembly line is incorrect, since `tickLast` is not shifted left as described.

Recommendation: Update the comments above the function to the following:

```
// s = (lastUpdateTime << 224) | tickLast
```

Ekubo: Fixed in commit 8d3726a.

Auditor: Verified.

3.6 Gas Optimizations

3.6.1 `createTwammPoolState()` bit shifting can be simplified

Description: The `createTwammPoolState()` function has the following implementation:

```
function createTwammPoolState(uint32 _lastVirtualOrderExecutionTime, uint112 _saleRateToken0,
    uint112 _saleRateToken1)
    pure
    returns (TwammPoolState s)
{
    assembly ("memory-safe") {
        // s = (lastVirtualOrderExecutionTime) | (saleRateToken0 << 32) | (saleRateToken1 << 144)
        s := or(
            or(and(_lastVirtualOrderExecutionTime, 0xffffffff), shl(32, shr(144, shl(144,
                _saleRateToken0)))),
            shl(144, shr(144, shl(144, _saleRateToken1)))
        )
    }
}
```

The `_saleRateToken0` and `_saleRateToken1` values are masked by shifting left and then right by 144 bits, which ensures dirty upper bits are cleared. This is followed by a final shift to position the value for packing. The last two shifts can be combined without changing the behavior.

Recommendation: To remove the unnecessary third shift, consider changing `shl(32, shr(144, shl(144, _saleRateToken0)))` to `shr(112, shl(144, _saleRateToken0))`, and changing `shl(144, shr(144, shl(144, _saleRateToken1)))` to `shl(144, _saleRateToken1)`.

Ekubo: Fixed in commit f731e7b.

Auditor: Verified.

3.6.2 `PoolKey` memory is allocated twice

Description: In both the TWAMM and MEVCapture contracts, the `locked_6416899205()` function manually allocates memory for the `poolKey` struct using inline assembly. This is unnecessary, as Solidity already allocates memory for the `poolKey` when the variable is declared.

Recommendation: Consider removing the manual memory allocation for the `poolKey`.

Ekubo: Addressed in commit 28ddd70 and commit cd7fc65.

Auditor: Verified.

3.6.3 `accumulatePoolFees()` can avoid no-op `lock()` calls

Description: The `accumulatePoolFees()` function is the only function that can trigger the `locked_6416899205()` callback in MEVCapture. The `locked_6416899205()` function is a no-op when `lastUpdateTime == currentTime`. One gas optimization would be to check this condition earlier in `accumulatePoolFees()`, so the `lock()` and callback are skipped entirely when no action is needed.

This change would save gas when a pool is interacted with multiple times in the same block, since `accumulatePoolFees()` is called by functions like `beforeCollectFees()` and `beforeUpdatePosition()`.

Recommendation: Move the `lastUpdateTime == currentTime` check earlier into `accumulatePoolFees()` to avoid unnecessary `lock()` calls when the callback would be a no-op.

Ekubo: Addressed in commit 2598de4.

Auditor: Verified.

4 Appendix

4.1 Known Risks and Behaviors

This review was conducted in advance of a Code4rena audit contest. To assist with the contest, a list of known risks and known behavior has been compiled to be shared with contest participants ahead of time. Contestants may find this list useful to consider, especially to see whether any of the issues can be escalated or exploited in ways not already anticipated.

- If a `BaseNonfungibleToken` NFT is burned, it can be re-minted by the original minter. This behavior is already documented in the code comments. While this isn't an issue within the codebase itself, integrators should be aware of this behavior and account for it accordingly.
- Some EVM chains have “hybrid” ERC20/native tokens, which are ERC20 tokens that mirror the chain's native token while the native balance still exists independently. Well-known examples include Celo, Polygon, and zkSync Era.

With these chains, a native `msg.value` transfer within the `startPayments()` + `completePayments()` flow on the hybrid ERC20 would be double-counted toward both the native balance of `address(0)` and the hybrid ERC20 balance. In the case of Celo, both the native and ERC20 balances can be withdrawn as native CELO, so an attacker can steal all native tokens. This is known behavior and the contracts cannot be deployed as-is with native token support on Celo. On Polygon, the hybrid ERC20 requires `msg.value` to be sent during a `transfer()`, so it cannot be withdrawn via `withdraw()`. The same is true on zkSync Era, since the hybrid token lacks a `transfer()` function entirely. So for both Polygon and zkSync Era, although an attacker can inflate their hybrid ERC20 balance arbitrarily, this is only exploitable if someone else provides liquidity using those tokens, which can be avoided if users only use the `address(0)` native token. This has been acknowledged as an accepted risk.

- The `FlashAccountant` uses transient storage slots that are derived from hashing strings such as `"FlashAccountant##_DEBT_LOCKER_TOKEN_ADDRESS_OFFSET"`. These transient slots are combined with `token` and possibly `id` offsets, for example, the debt of a `token` in a specific `id` is stored in the transient slot `add(_DEBT_LOCKER_TOKEN_ADDRESS_OFFSET, add(shl(160, id), token))`. In theory, an arbitrary `token` and `id` could result in this slot colliding with any other slot. However, this is not an issue in practice, since `id` can't be arbitrary. It is simply incremented on each call to `lock()`, so it will always remain relatively small. Also, none of the root transient slot hashes share the same upper 96 bits.
- The `Incentives` contract tracks claims using bitmap storage slots derived from `dropId` and `index`. At first glance, it may seem possible to construct a Merkle tree that includes a carefully chosen `index` whose bitmap slot collides with storage from another drop. However, since `dropId` is computed from the Merkle root, which itself depends on the tree's contents, this leads to circular logic. An attacker can't know which `index` to use for a collision until they know the `dropId`, which already depends on the indices in the tree. So this idea does not appear to be exploitable.
- The value returned by `_getPoolFeesPerLiquidityInside()` should not be interpreted as an absolute quantity with meaningful units. What matters is that the relative growth of this function is consistent once the endpoints of a tick range are initialized. To see why this is, notice that `_updateTick()` always sets a newly-initialized tick's "fees per liquidity outside" fields to 1 wei, regardless of whether the tick lies above or below the current tick. This means the actual value of `_getPoolFeesPerLiquidityInside()` for a tick range can vary depending on when each endpoint tick was first initialized. However, once initialized, the tick crossing logic does ensure that relative growth within the range is consistent, which is sufficient for the protocol's needs.
- The transient `coreBalance` within the `TokenWrapper` is never decremented. In theory, a user could call `startPayments()`, transfer wrapped tokens to `CORE`, then `withdraw()` the tokens, and finally call `completePayments()`. This would result in no net debt change, but the `coreBalance` would be incremented due to the first transfer. While this can grow `coreBalance`, an overflow does not appear feasible, and even if it were, it does not clearly lead to an exploit.
- TWAMM pools could become stuck if `_executeVirtualOrdersFromWithinLock()` reverts, as this function runs before all state-changing operations in the pool. No concrete way to trigger such a revert was found during the audit. One possibility that was explored involved pushing the pool's tick or sqrtPrice to the min or max boundary, then queuing a TWAMM swap that would further push the price toward the boundary. While it

is possible to push the current tick to `MIN_TICK - 1`, even in this case the resulting sqrtRatio remains within bounds, and no revert occurs.

- The `RevenueBuybacks` contract allows any token address to be passed to `roll()`, which reads from the storage slot equal to the token value. This introduces a theoretical risk of collision with the contract's `Ownable` storage. The `Ownable` logic uses `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffff74873927` for the `owner` and `keccak256(_HANDOVER_SLOT_SEED, msg.sender)` for ownership handover. Since token addresses are only 20 bytes, collisions with the `owner` slot are impossible. However it is theoretically possible to attempt to mine an address such that its `requestOwnershipHandover()` slot lands in a 20-byte address range. This would be computationally very expensive and unlikely to be useful since the equivalent `token` wouldn't be a real token anyway.
- In the TWAMM extension, the following code calculates the net amount of tokens to be paid when creating or updating an order:

  ```
  // the amount required for executing at the next sale rate for the remaining duration of
      the order
  uint256 amountRequired =
      computeAmountFromSaleRate({saleRate: saleRateNext, duration: durationRemaining, roundUp
          : true});

  // subtract the remaining sell amount to get the delta
  int256 amountDelta;

  uint256 remainingSellAmount =
      computeAmountFromSaleRate({saleRate: saleRate, duration: durationRemaining, roundUp:
          true});

  assembly ("memory-safe") {
      amountDelta := sub(amountRequired, remainingSellAmount)
  }
  ```

  In this code, `amountRequired` represents the total owed at the new sale rate, while `remainingSellAmount` reflects the value already paid under the previous rate. Subtracting the two yields the net amount due. The `remainingSellAmount` is rounded up. This favors the user and doesn't follow the typical heuristic of rounding in favor of the protocol. No explicit issue was found relating to this behavior, and all other rounding locations favor the protocol and round against the user.
- Some assembly blocks are marked as `memory-safe` even though they technically violate `memory-safe` constraints. For example, consider this code:

  ```
  assembly ("memory-safe") {
      // cast sig "updateDebt()"
      mstore(0, 0x17c5da6a)
      mstore(32, shl(128, delta))

      if iszero(call(gas(), accountant, 0, 28, 20, 0, 0)) {
          returndatacopy(0, 0, returndatasize())
  ```

```
            revert(0, returndatasize())
        }
    }
```

This is a violation as can be seen from this note in the Solidity docs:

> Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates. As an example, the following assembly snippet is not memory safe, because the value of `returndatasize()` may exceed the 64 byte scratch space:
>
> ```
> assembly {
>     returndatacopy(0, 0, returndatasize())
>     revert(0, returndatasize())
> }
> ```

Because the `memory-safe` annotation tells the compiler which optimizations are safe to perform, an assembly block that is incorrectly marked is technically undefined behavior. However, it's unlikely that these code snippet would cause unexpected behavior since they immediately revert.