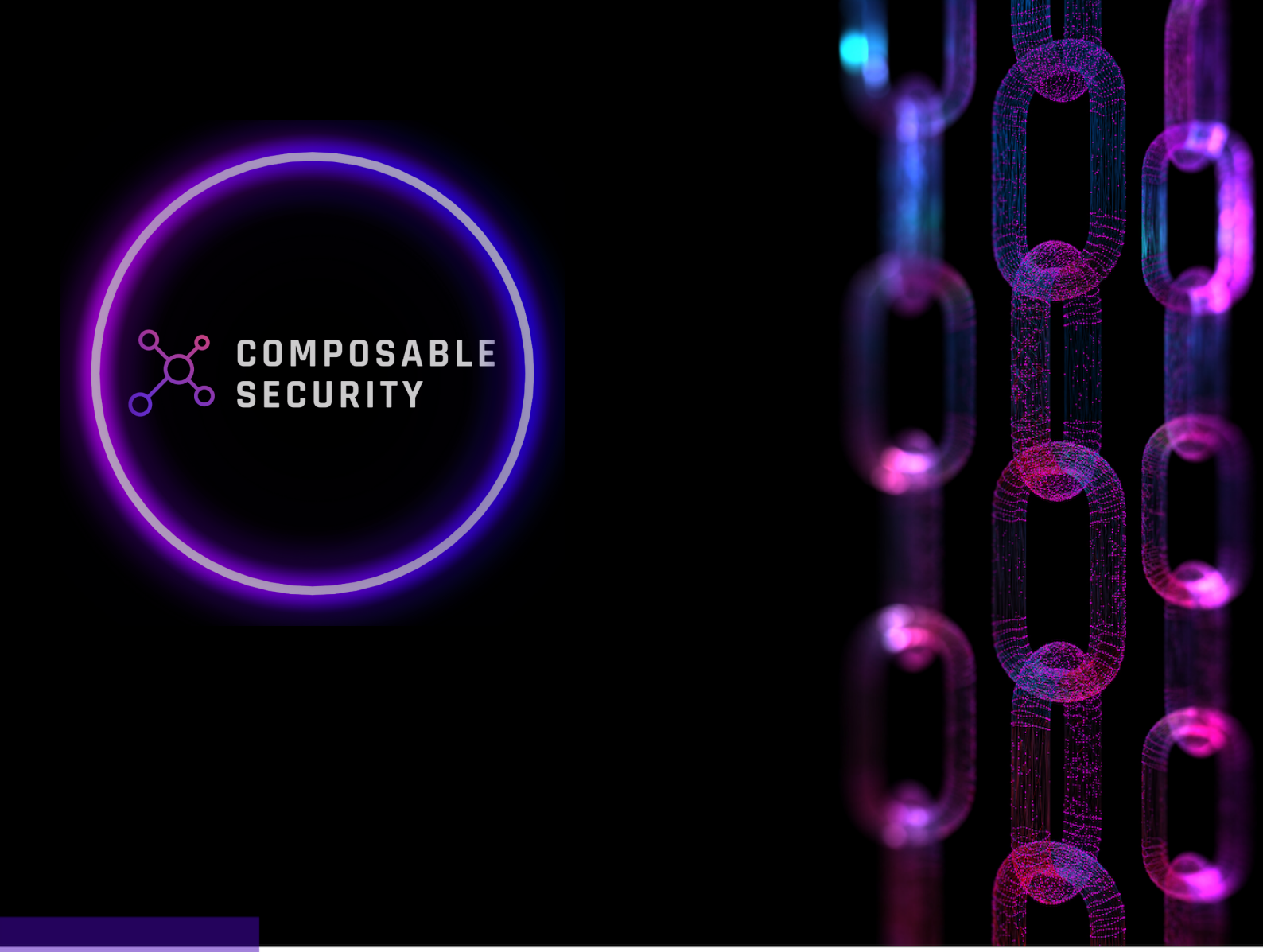


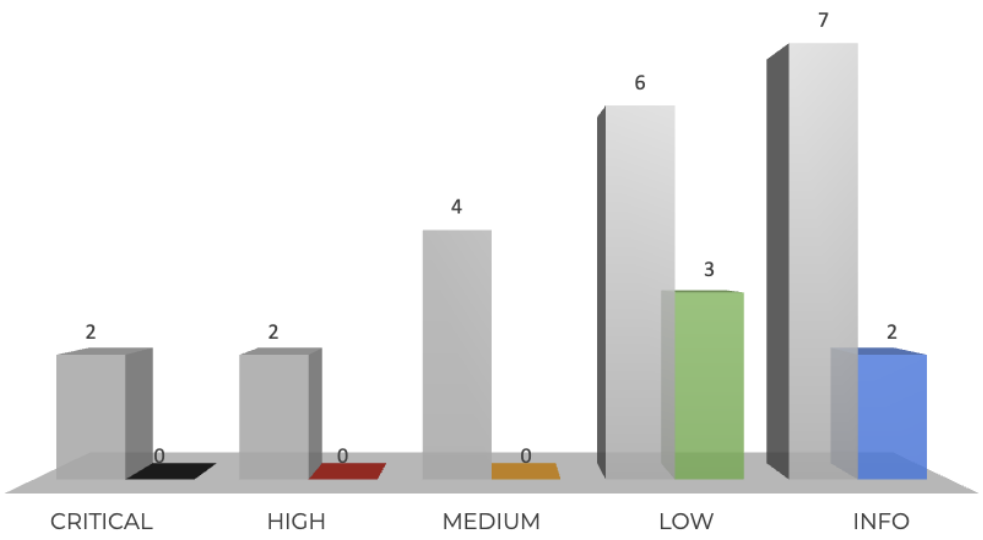
REPORT

Smart contract security review for Arbiter

Prepared by: Composable Security
Report ID: ARB-d7f31318
Test time period: 2025-01-27 - 2025-02-11
Retest time period: 2025-03-12 - 2025-03-28
Report date: 2025-03-28
Version: 1.0
Visit: composable-security.com

Contents

1. Retest summary (2025-03-28)



The description of the current status for each retested vulnerability and recommendation has been added in its section.

1.1. Results

The **Composable Security** team was involved in a one-time iteration of verification whether the vulnerabilities detected during the tests (between 2025-01-27 and 2025-02-11) were removed correctly and no longer appear in the code.

The current status of detected issues is as follows:

- All **critical** vulnerabilities have been fully fixed.
- All **high** vulnerabilities have been been fully fixed.
- All **medium** vulnerabilities have been been fully fixed.
- **6** vulnerabilities with a **low** impact on risk were handled as follows:
  - 2 have been acknowledged,
  - 3 have been fixed,
  - 1 has been partially fixed.
- 7 security **recommendations** were handled as follows:
  - 5 have been implemented,
  - 1 has been partially implemented,
  - 1 has not been implemented.

1.2. Scope

The retest scope included the same contracts, on a different commit in the same repository.

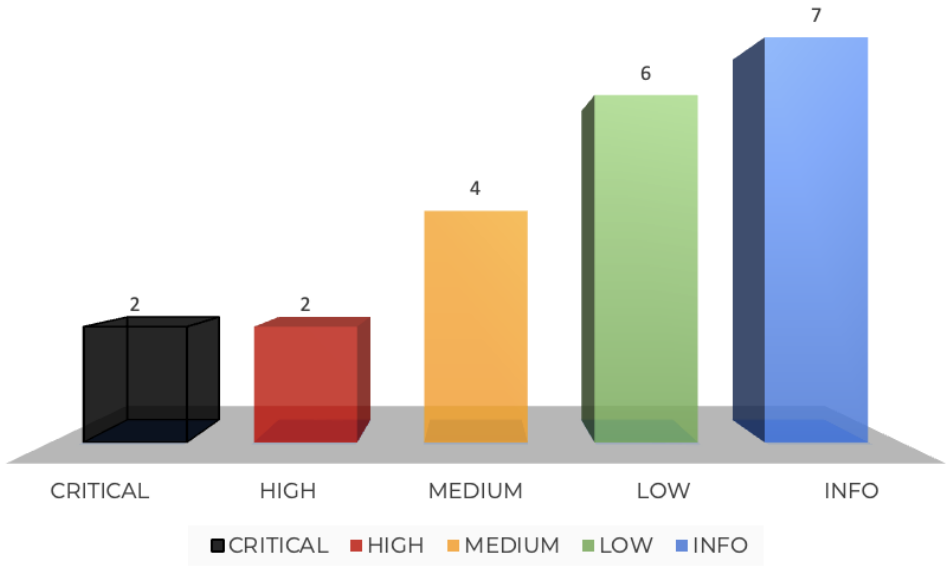
GitHub repository: https://github.com/ArbiterFinance/arbiter-contracts
CommitID: 8afdf775cb407892387fde3f538e4338fac254a6

2. Current findings status

ID	Severity	Vulnerability	Status
ARB-d7f31318-C01	CRITICAL	Reward growth updated for invalid tick	FIXED
ARB-d7f31318-C02	CRITICAL	Invalid calculations of reward per liquidity	FIXED
ARB-d7f31318-H01	HIGH	Loss of auction fees	FIXED
ARB-d7f31318-H02	HIGH	Invalid pool tick stored before initialization	FIXED
ARB-d7f31318-M01	MEDIUM	Deprecated values in slot0 and slot1	FIXED
ARB-d7f31318-M02	MEDIUM	Wrong mask value	FIXED
ARB-d7f31318-M03	MEDIUM	Invalid active liquidity update	FIXED
ARB-d7f31318-M04	MEDIUM	DoS on swaps due to division by zero	FIXED
ARB-d7f31318-L01	LOW	Last active tick never updated	FIXED
ARB-d7f31318-L02	LOW	Invalid currency returned	ACKNOWLEDGED
ARB-d7f31318-L03	LOW	Winner steals whole swap amount	ACKNOWLEDGED
ARB-d7f31318-L04	LOW	Inability to burn liquidity position	PARTIALLY FIXED
ARB-d7f31318-L05	LOW	Bypassing limits when changing strategies	FIXED
ARB-d7f31318-L06	LOW	Unnecessary cross for uninitialized ticks	FIXED

ID	Severity	Recommendation	Status
ARB-d7f31318-R01	INFO	Consider using named mappings	NOT IMPLEMENTED
ARB-d7f31318-R02	INFO	Remove unnecessary code	NOT IMPLEMENTED
ARB-d7f31318-R03	INFO	Consider using newest Solidity version	IMPLEMENTED
ARB-d7f31318-R04	INFO	Remove TODOs	IMPLEMENTED
ARB-d7f31318-R05	INFO	Use custom errors	IMPLEMENTED
ARB-d7f31318-R06	INFO	Fix comments	IMPLEMENTED
ARB-d7f31318-R07	INFO	Emit events for important state changes	IMPLEMENTED

3. Security review summary (2025-02-11)



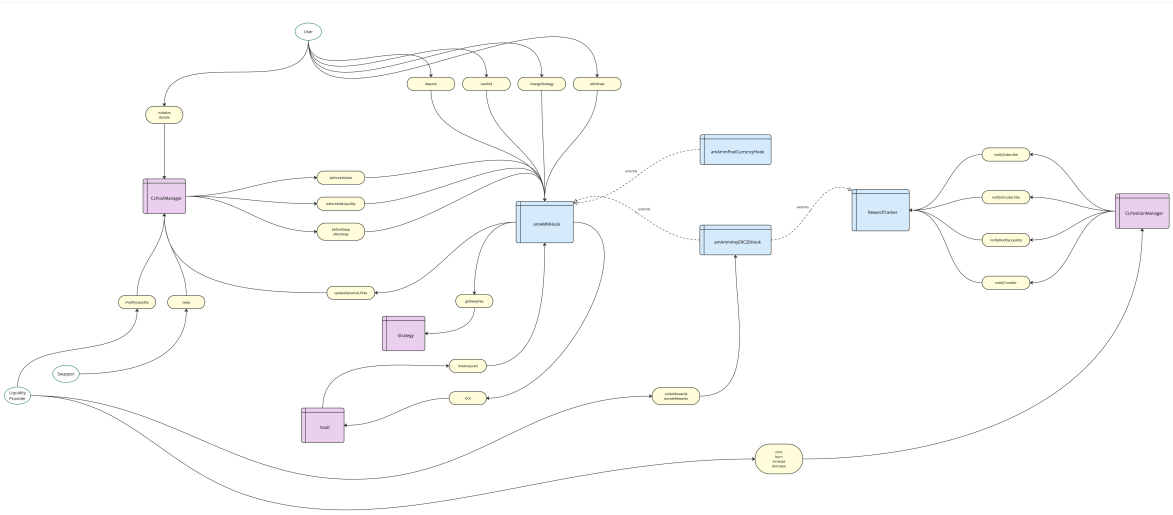
3.1. Client project

The **Arbiter amAMM** project introduces an innovative mechanism to enhance Automated Market Makers (AMMs) by integrating a continuous auction system for managing and collecting swap fees.

At its core is the Harberger Lease Hook, which facilitates ongoing auctions where participants can bid for the right to manage a specific AMM pool's fees. This auction operates continuously, allowing users to outbid the current manager at any time, with the new manager assuming control from the next block onward. Bids can be placed using the pool's trading tokens or alternative tokens, and the proceeds are distributed to the pool's liquidity providers (LPs).

Notably, when a pool's trading token is used for bidding, rewards are seamlessly distributed to LPs without requiring additional actions.

Conversely, if another token is used, LPs need to subscribe to the hook via the PositionManager to be eligible for rewards, which are then allocated proportionally based on their active participation.



3.2. Results

The **Arbiter** engaged Composable Security to review the security of **Arbiter**. Composable Security conducted this assessment over 1 person-week with 2 engineers.

The summary of findings is as follows:

- 2 **critical** risk impact vulnerabilities were identified. Their potential consequence is:
  - An attacker can steal the whole amount of deposited rent tokens. (ARB-d7f31318-C01 and ARB-d7f31318-C02).
- 2 vulnerabilities with a **high** impact on risk were identified.
- 4 vulnerabilities with a **medium** impact on risk were identified.
- 6 vulnerabilities with a **low** impact on risk were identified.
- 7 **recommendations** have been proposed that can improve overall security and help implement best practice.
- The most important issues detected concern the calculation of rewards accrued by the liquidity providers.
- Many critical and high impact vulnerabilities were discovered in the limited testing time-frame, indicating room for code improvement and internal peer reviews.

Composable Security recommends that **Arbiter** complete the following:

- Address all reported issues.
- Extend unit tests with scenarios that cover detected vulnerabilities where possible.
- Consider whether the detected vulnerabilities may exist in other places (or ongoing projects) that have not been detected during engagement.
- Review dependencies and upgrade to the latest versions.

3.3. Centralization Risk

The current system implementation is not fully decentralized, allowing critical operations (e.g., changes of contracts' parameters) to be performed without additional security measures such as timelocks, granular permissions or multi-step processes.

This poses risks to the project's security, as it places a substantial amount of trust in the Owner role. It is crucial to ensure the highest level of protection for the private keys associated with this role.

Even if the project plans to execute important changes via a multi-signature (multi-sig) setup, this does not fully address the underlying issue. Operations can still be introduced suddenly without warning users, and the controlling parties retain complete control over user funds. Even though the code does not indicate any malicious intent, the users should be aware of their dependency.

Recommendation:

- To mitigate the risks associated with single points of failure or potential compromises, appropriate security requirements and procedures should be established to limit the impact in the event of loss of access or key compromise.
- Consider using `TimelockController` contract to delay changes of important parameters that can influence users's funds.
- Use multi-sig wallets to control the protocol contracts and require to initiate any critical updated by those wallets.

References:

1. Secure Private Key Management for DApps
2. SCSVS G2: Policies and procedures
3. SCSVS G1: Architecture, design and threat modeling

3.4. Scope

The scope of the tests included selected contracts from the following repository.

GitHub repository: https://github.com/ArbiterFinance/arbiter-contracts
CommitID: d7f313181fc2f12ce3ac33b2c28410b532d58a56

The detailed scope of tests can be found in Agreed scope of tests.

4. Project details

4.1. Projects goal

The Composable Security team focused during this audit on the following:

- Perform a tailored threat analysis.
- Ensure that smart contract code is written according to security best practices.
- Identify security issues and potential threats both for **Arbiter** and their users.
- The secondary goal is to improve code clarity and optimize code where possible.

4.2. Agreed scope of tests

The subjects of the test were selected contracts from the **Arbiter** repository.

GitHub repository:
https://github.com/ArbiterFinance/arbiter-contracts
CommitID: d7f313181fc2f12ce3ac33b2c28410b532d58a56

Files in scope:

```
.
└── ./src
    ├── ./src/ArbiterAmAmmAnyERC20Hook.sol
    ├── ./src/ArbiterAmAmmBaseHook.sol
    ├── ./src/ArbiterAmAmmPoolCurrencyHook.sol
    ├── ./src/RewardTracker.sol
    ├── ./src/libraries
    │   ├── ./src/libraries/PoolExtension.sol
    │   └── ./src/libraries/PositionExtension.sol
    └── ./src/types
        ├── ./src/types/AuctionSlot0.sol
        └── ./src/types/AuctionSlot1.sol
```

Documentation:

- GitHub Docs
- Arbiter amAMM - HackMD

4.3. Threat analysis

This section summarizes the potential threats that were identified during initial threat modeling performed before the audit. The tests were focused, but not limited to, finding security

issues that could be exploited to achieve these threats.

Key assets that require protection:

- Deposited tokens.
- Pool tokens.

Potential attackers goals:

- Theft of deposited tokens.
- Reward amount manipulation.
- Lock user tokens in the contract.
- Block the contract, so that others cannot use it.

Potential scenarios to achieve the indicated attacker's goals:

- Using not updated or outdated data from the pool.
- Invalid update of fees for tick ranges.
- Overwriting fee value when changing the winner.
- Unauthorized strategy change to a reverting one.
- Use of wrong token as the fee.
- Direct call to notify functions by anonymous user.
- Cumulating rewards in actions being part of a batch.
- Influence or bypass the business logic of the system.
- Take advantage of arithmetic errors.
- Privilege escalation through incorrect access control to functions or poorly written modifiers.
- Existence of known vulnerabilities (e.g., front-running, re-entrancy).
- Design issues.
- Unintentional loss of the ability to govern the system.

4.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the **Arbiter** development team to thoroughly understand intentions and assumptions of the project.
- Initial threat modeling to identify key areas and focus on covering the most relevant scenarios based on real threats.
- Automatic tests using slither.
- Custom scripts (e.g. unit tests) to verify scenarios from initial threat modeling.
- **Manual review of the code.**

4.5. Disclaimer

Smart contract security review **IS NOT A SECURITY WARRANTY**.

During the tests, the Composable Security team makes every effort to detect any occurring problems and help to address them. However, it is not allowed to treat the report as a security certificate and assume that the project does not contain any vulnerabilities. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we encourage the implementation of security mechanisms at all stages of development and maintenance.

5. Vulnerabilities

[ARB-d7f31318-C01] Reward growth updated for invalid tick

CRITICAL FIXED

Retest (2025-03-20)

The vulnerability has been fixed as recommended.
CommitID: *286b16f5d24f6fa0d8efa5a488330fa22c11e139*

Affected files

- PoolExtension.sol#L179

Description

The `crossToActiveTick` function from `PoolExtension` library is used to track changes of the current tick in the pool and update the crossed ticks to correctly account for the rewards belonging to the tick ranges. Its logic is based on the PancakeSwap's library.

However, the function `PoolExtension.crossTick` is being called **not** for the next tick that is about to be crossed but for the current tick, which is also misaligned when going left (e.g. tick being crossed is -1 when the tick spacing is 60). This leads to a situation where not only wrong ticks are being updated but a different ones are crossed depending on the direction.

That is, when starting from tick 0 and going left, first tick 0 is crossed, then tick -1 is crossed, then the "next initialized tick" - 1 is crossed, etc., but when going back, the ticks being crossed are "next initialized tick" and 0. The outside rewards are not symmetrically updated, leading to invalid rewards per liquidity being calculated by the hook.

Attack scenario

The attackers might take the following steps in turn:

① Initially the pool has current tick equal to 0, tick spacing 60 and some liquidity added in a symmetric range (-120, 120). The outside reward for tick -120 is set to 0, because no rewards exist yet.
② A bidder had won the auction with 10 token per block.
③ Nothing happens for 20 blocks. The reward becomes 200 tokens.
④ Attacker executes a swap that moves the tick left to -120. The hook crosses ticks 0 and -1.

⑤ Attacker adds some liquidity for range (-120, -60). No outside reward is updated because there was already liquidity added for -120 tick. The last used reward is equal to current cumulative reward.
⑥ Attacker executes a swap in a different direction that moves the tick to -80. The hook crosses tick -120 and sets its outside reward to current cumulative reward.
⑦ Attacker removes liquidity and accrues rewards. The current tick is in the middle of the range so the reward is current cumulative reward minus outside rewards of both ticks, which is equal to 0. However, when one subtracts the last used reward, an unchecked underflow happens and the reward becomes a huge number.

By manipulating the amount liquidity added in step 5, the attacker can make the reward equal to the amount of deposited rent tokens in the hook.

Result of the attack: An attacker can steal the whole amount of deposited rent tokens.

Recommendation

Call the `crossTick` function for the `nextTick` variable but only if it would be crossed in the current loop iteration. It can be checked with condition `(activeTick < nextTick) == goingLeft`:

```
if (initialized) {
    if ((activeTick < nextTick) == goingLeft) {
      liquidityNet = PoolExtension.crossTick(
        self,
        nextTick,
        self.rewardsPerLiquidityCumulativeX128
    );
  }
}
```

References

1. SCSVS G4: Business logic

[ARB-d7f31318-C02] Invalid calculations of reward per liquidity

CRITICAL **FIXED**

Retest (2025-03-12)

The vulnerability has been fixed as recommended.
CommitID: *57db8e7831f76bf2da3d76018b5eb5f97084a6f1*

Affected files

- PoolExtension.sol#L39-L62

Description

The provided calculations within `getRewardsPerLiquidityInsideX128` function omit one of the necessary "outside" reward values (or subtract them in the wrong order). In the extreme cases it ends up with an unchecked underflows and huge reward value.

They do not match the standard method for computing fee/reward growth inside a tick range.

1. Current tick below tickLower

Expected: `rewardPerLiquidityInside = O(lower)- O(upper)`

In the code:

```
if (self.tick < tickLower) {
    return
        self.ticks[tickLower].rewardsPerLiquidityOutsideX128 -
        self.rewardsPerLiquidityCumulativeX128;
}
```

It does not reference `tickUpper` and will typically be negative (since the global cumulative growth G is usually larger than `O(lower)`). This is not the intended calculation.

2. Current tick at or above tickUpper

Expected: `rewardPerLiquidityInside = O(upper)- O(lower)`

In the code:

```
if (self.tick >= tickUpper) {
    return
        self.rewardsPerLiquidityCumulativeX128 -
        self.ticks[tickUpper].rewardsPerLiquidityOutsideX128;
 }
```

Calculations ignore the lower tick's stored value.

3. Current tick inside the range (between tickLower and tickUpper)

Expected: `rewardPerLiquidityInside = G - O(lower)- O(upper)`

In the code:

```
return
    self.rewardsPerLiquidityCumulativeX128 -
    self.ticks[tickUpper].rewardsPerLiquidityOutsideX128 -
    self.ticks[tickLower].rewardsPerLiquidityOutsideX128;
}
```

This case is correctly calculated.

Attack scenario

The attackers might take the following steps in turn:

① There is a pool without any rewards distributed yet (`rewardsPerLiquidityCumulativeX128 == 0`).
② The attacker adds some liquidity in a range which has greater ticks than the current tick (e.g. `(153600, 153660)`). The outside rewards are not set, because ticks are greater than current tick.
③ The last rewards per liquidity is set to 0 - 0 = 0.
④ The attacker waits some time until cumulative rewards get accumulated.
⑤ The attacker burns the liquidity position. The current reward per liquidity is 0 - "current cumulative liquidity" which is equal to a huge number because of the underflow.
⑥ The attacker collects the reward and steals deposited rent tokens.

By manipulating the amount liquidity added in step 2, the attacker can make the reward equal to the amount of deposited rent tokens in the hook.

Result: An attacker can steal the whole amount of deposited rent tokens.

Recommendation

Correct the calculations according to:

```
if (self.tick < tickLower) {
    return self.ticks[tickLower].rewardsPerLiquidityOutsideX128 - self.
        ticks[tickUpper].rewardsPerLiquidityOutsideX128;
}
if (self.tick >= tickUpper) {
    return self.ticks[tickUpper].rewardsPerLiquidityOutsideX128 - self.
        ticks[tickLower].rewardsPerLiquidityOutsideX128;
}
return
    self.rewardsPerLiquidityCumulativeX128 -
    self.ticks[tickUpper].rewardsPerLiquidityOutsideX128 -
```

```
    self.ticks[tickLower].rewardsPerLiquidityOutsideX128;
```

References

1. SCSVS G7: Arithmetic

[ARB-d7f31318-H01] Loss of auction fees

HIGH FIXED

Retest (2025-03-12)

The vulnerability has been fixed as recommended.
CommitID: *3e202f8f070c23b6c13e49529dc4b042d1d9a8d1*

Affected files

- ArbiterAmAmmBaseHook.sol#L393

Description

The auction mechanism allows to smoothly change the winner when they overbid the current one. When the current winner still have some remaining rent, it is returned with the proportional fee and the collected fee (earned by the protocol) is stored in `auctionFees` mapping.

```
uint128 feeRefund = uint128(
    (uint256(prevAuctionFee.feeLocked) * remainingRent) /
        prevAuctionFee.initialRemainingRent
);
uint128 collectedFee = prevAuctionFee.feeLocked - feeRefund;

deposits[winners[poolId]][currency] +=
    slot1.remainingRent() +
    feeRefund;
auctionFees[poolId] = AuctionFee(0, 0, collectedFee);
```

However, if there is already a collected fee stored from previous overbids, this code overwrites it with the new value from the current overbid. In effect, if the auction owner does not call the fee collection function between overbids, the accumulated collected fees are lost permanently.

Attack scenario

The attackers might take the following steps in turn:

① An attacker repeatedly calls the overbid function, forcing the contract to perform multiple overbid operations in rapid succession.
② Each new overbid recalculates the fee refund and the resulting collected fee. The new calculated `collectedFee` overwrites the value in the `auctionFees` mapping, effectively erasing any fee that was previously accrued.
③ As a result, when the owner eventually calls `collectAuctionFees`, only the fee from the last overbid is available for collection, while the fees that should have been accumulated from earlier overbids are permanently lost.

Result of the attack: The protocol loses a significant portion of the auction fees that should have accumulated over time.

Recommendation

Modify the fee refund logic in the overbid function so that instead of overwriting the existing fee data, the new `collectedFee` is added to any previously stored value.

References

1. SCSVS G4: Business logic

[ARB-d7f31318-H02] Invalid pool tick stored before initialization

HIGH FIXED

Retest (2025-03-12)

The vulnerability has been fixed as recommended.
CommitID: *fa9b5adf3bec2cd2c59d0d5ef6300f1eb7da69d2*

Affected files

- ArbiterAmAmmBaseHook.sol#L118
- ArbiterAmAmmAnyERC20Hook.sol#L61
- ArbiterAmAmmAnyERC20Hook.sol#L71

Description

When the pool is initialized, the `beforeInitialize` hook function is called. One of its operation is storing the current tick, that is the tick that was used to initiate the pool.

However, before the pool is initialized, the tick is not set and the `getSlot0` function will always return 0, even if the initial price is different than 1:1.

Later, many operation in `RewardTracker` contract would use invalid value of the current tick:

- the `getRewardsPerLiquidityInsideX128` function would return invalid reward per liquidity,
- the `crossToActiveTick` function would make a lot of unnecessary while loop iterations,
- the `updateTick` function would incorrectly set outside rewards for updated ticks.

Result: Invalid rewards calculated by `RewardTracker` contract.

Recommendation

Access the current tick in `afterInitialize` hook function.

References

1. SCSVS C9: Uniswap V4 Hook

[ARB-d7f31318-M01] Deprecated values in slot0 and slot1

MEDIUM FIXED

Retest (2025-03-20)

The vulnerability has been fixed as recommended.
However, only `slot1` has been retrieved again from the storage, because the values in `slot0`, which are later used (i.e. `auctionFee`) are not changed.
CommitID: *96aa53e4c561c1280ff99fcc54104f4a8c7b7ee2*

Affected files

- ArbiterAmAmmBaseHook.sol#L380

Description

The `overbid` function is used to take part in an auction that selects the strategist for the next blocks. It reads pool slots to memory variables and later update it.

```
AuctionSlot0 slot0 = poolSlot0[poolId];
```

```
AuctionSlot1 slot1 = poolSlot1[poolId];
```

Later, the function calls another function `_payRentAndChangeStrategyIfNeeded` which also uses and updates these slots. However, these slots are not re-read from storage after the function call, but still use the values that were read before the `_payRentAndChangeStrategyIfNeeded` function was called.

One specific variable kept in that slot is `remainingRent` which represents the amount of rent token that is still remaining for the current winner.

Even though the `_payRentAndChangeStrategyIfNeeded` function can clear it, the `overbid` function still uses old value leading to a situation where the winner can receive their rent back, effectively stealing it.

Attack scenario

The attackers might take the following steps in turn:

① Win the current auction to be strategist for some blocks.
② If there are no operations (swaps, liquidity modifications or overbids) for some time, the attacker calls the `overbid` function to overbid themselves.
③ The hook calls `_payRentAndChangeStrategyIfNeeded` to distribute rewards.
④ The hook checks what is the remaining rent to calculate the refund. The remaining rent is the initial amount and the hook returns the whole amount of rent back to the winner (attacker).

If the attacker is also the liquidity provider, they can effectively steal some tokens in step 3.

Result of the attack: Theft of the rent amount that has already been spent on the past blocks and part of the rewards.

Recommendation

Update the values of `slot0` and `slot1` memory variables after the `_payRentAndChangeStrategyIfNeeded` function:

```
_payRentAndChangeStrategyIfNeeded(key);

slot0 = poolSlot0[poolId];
slot1 = poolSlot1[poolId];
```

References

1. SCSVS G4: Business Logic

[ARB-d7f31318-M02] Wrong mask value

MEDIUM FIXED

Retest (2025-03-20)

The vulnerability has been fixed as recommended.
CommitID: *cbc600b3fa83e0d496aa979336e1d79181890ac1*

Affected files

- AuctionSlot1.sol#L20

Description

The value of the `MASK_96_BITS` variable used to extract the amount of rent paid per block from the data structure is incorrect.

```
uint96 internal constant MASK_96_BITS = 0xFFFFFFFFFFFFFFFFFFFF;
```

A mask for 96 bits is (1«96)-1. In hexadecimal, it is 24 F's (24 × 4 = 96) and not 20 F's.

Result: Invalid amount of rent paid per block extracted and invalid parameters stored in packed slot.

Recommendation

The 96-bit mask needs to be updated to properly mask 96 bits.

```
uint96 internal constant MASK_96_BITS = 0xFFFFFFFFFFFFFFFFFFFFFFFF;
```

References

1. SCSVS G4: Business logic

[ARB-d7f31318-M03] Invalid active liquidity update

MEDIUM FIXED

Retest (2025-03-20)

The vulnerability has been fixed as recommended.
CommitID: *c60364a98925ba263f3f343ceb9e5787f45ea834*

Affected files

- PoolExtension.sol#L147

Description

In many liquidity pool designs (for example, Uniswap V3), a liquidity position is considered active for ticks in the range [tickLower, tickUpper). This means that the lower bound is usually inclusive while the upper bound is exclusive.

Current implementation does not take into account case where `params.tickLower == self.tick`.

```
// update the active liquidity
if (params.tickLower < self.tick && self.tick < params.tickUpper) {
    self.liquidity = LiquidityMath.addDelta(
    self.liquidity,
    liquidityDelta
    );
}
```

Vulnerable scenario

The following steps lead to the described result:

① There is liquidity amount of X between ticks -120 and 120.
② During the swap the fee is divided by X to calculate fee growth and the current tick is -120.
③ The user adds liquidity amount of Y between ticks -120 and 60.
④ The hook does not account it as the current liquidity.
⑤ The next swap fee is still divided by X and not X+Y to calculate fee growth, making the growth incorrectly high.

Result: The added liquidity is not included in active liquidity, incorrectly increasing the fee growth for current liquidity.

Recommendation

The active liquidity condition should include the lower bound (using <=) instead of strict <.

```
if (params.tickLower <= self.tick && self.tick < params.tickUpper) {
```

References

1. SCSVS G7: Arithmetic

[ARB-d7f31318-M04] DoS on swaps due to division by zero

MEDIUM FIXED

Retest (2025-03-20)

The vulnerability has been fixed. However, after the fix, the pool cannot be used if there is no liquidity added through the PositionManager.
CommitIDs:
c60364a98925ba263f3f343ceb9e5787f45ea834,
8afdf775cb407892387fde3f538e4338fac254a6

Affected files

- PoolExtension.sol#L97

Description

In order to collect rent fees by liquidity providers in pools with a custom ERC20 as the rent currency, they have to add liquidity through the position manager, because the `RewardTracker` contract requires it.

However, the hook does not detect situations when some liquidity has been added directly. If there is some liquidity added, but none through the position manager, all swaps will revert because there liquidity amount stored in `RewardTracker` contract is zero and the contract tries to divide fee by zero.

```
95 self.rewardsPerLiquidityCumulativeX128 +=
96             (uint256(rewardsAmount) << 128) /
97             self.liquidity;
```

Vulnerable scenario

The following steps lead to the described result:

① The liquidity provider adds liquidity directly, without using the position manager.
② The user makes a swap.
③ The hook reverts in `distributeRewards` function due to division by zero.

Result: Denial of service on swaps.

Recommendation

Detect a case when there is no liquidity tracked and store fee in a separate variable to be later distributed or collected by the owner.

References

1. SCSVS G8: Denial of service

[ARB-d7f31318-L01] Last active tick never updated

LOW FIXED

Retest (2025-03-20)

The vulnerability has been fixed as recommended.
CommitID: *211b84adcb1424cb5531f7cb552e582b02a3dce1*

Affected files

- ArbiterAmAmmBaseHook.sol#L233
- ArbiterAmAmmAnyERC20Hook.sol#L87

Description

The contract uses a `lastActiveTick` (stored in the `AuctionSlot0`) as a signal to decide to call the rent-payment and strategy–change logic after swap, but that “last tick” isn’t updated when the rent is paid.

After a swap triggers a change (because the current tick is not equal to the stored last active tick), the function `afterSwap` checks whether the current tick differs from the stored last active tick.

```
function afterSwap(
    address,
    PoolKey calldata key,
    ICLPoolManager.SwapParams calldata,
    BalanceDelta,
    bytes calldata
  ) external virtual override poolManagerOnly returns (bytes4, int128) {
    PoolId poolId = key.toId();
    (, int24 tick, , ) = poolManager.getSlot0(poolId);

    AuctionSlot0 slot0 = poolSlot0[poolId];
```

```
    if (tick != slot0.lastActiveTick()) {
        _payRentAndChangeStrategyIfNeeded(key);
    }

    return (this.afterSwap.selector, 0);
    }
```

However, inside `_payRentAndChangeStrategyIfNeeded` no update is made to the `lastActiveTick` value. This means that once the tick changes relative to the initial value (set in `beforeInitialize`), subsequent swaps (even if they occur with the same price/tick as the one that first triggered the rent payment) will continue to see a difference between the current tick and the stale `lastActiveTick`.

Result: The rent–payment (and potential strategy change) logic could be triggered repeatedly when it shouldn't be increasing the cost of swaps.

Recommendation

Update the stored `lastActiveTick` after processing a rent payment.

References

1. SCSVS G4: Business logic

[ARB-d7f31318-L02] Invalid currency returned

LOW ACKNOWLEDGED

Retest (2025-03-20)

The team responded that this works as intended and do not plan to change the returned token.

Affected files

- ArbiterAmAmmBaseHook.sol#L485

Description

When user deposits some token to the hook, the token is sent to the vault and the hook mints its ERC6909 token equivalent. Later, when user withdraws the same deposited token, the hook burns its ERC6909 token equivalent and mints it again for the user.

```
483 if (data.withdrawAmount > 0) {
```

```
484     vault.burn(address(this), currencyWrapped, data.withdrawAmount);
485     vault.mint(data.sender, currencyWrapped, data.withdrawAmount);
486     vault.settle();
487 }
```

The user would rather want the deposited token itself and not its ERC6909 equivalent.

Vulnerable scenario

The following steps lead to the described result:

① The user deposits token X and transfers it out to the vault.
② The user withdraws token X.
③ The hook burns ERC6909 claim for token X and mints ERC6909 claim for token X to the user.

Result: The user receives ERC6909 equivalent of the deposited token instead of the token itself.

Recommendation

The hook should burn the ERC6909 token and take the deposited token itself on behalf of the user.

References

1. SCSVS G4: Business Logic

[ARB-d7f31318-L03] Winner steals whole swap amount

LOW ACKNOWLEDGED

Retest (2025-03-20)

The risk related to this vulnerability has been delegated to the swap router.
The team's comment: *It is a responsibility of swap executor to set* `amountOutMinimum` */* `amountInMaximum`*. Which if set, effectively shuts down the attack vector.*
Composable Security response: *PancakeSwap V4 allows to execute swaps directly on PoolManager which does not include the* `minAmountOut` *protection.*

Affected files

- ArbiterAmAmmBaseHook.sol#L174

Description

The `swapFee` can be set by Strategist anytime without proper limits on its maximum value. They can front-run the observed transaction and set the fee to 100

```
172 // Call strategy contract to get swap fee.
173 try
174     IArbiterFeeProvider(strategy).getSwapFee{
175         gas: 2 << slot0.strategyGasLimit()
176     }(sender, key, params, hookData)
177         returns (uint24 _fee) {
178     if (_fee <= 1e6) {
179         fee = _fee;
180     }
181 } catch {}
```

The stolen amount is not directly transferred to the strategist, only the winner's share is. However, if the strategist is also the main liquidity provider, the stolen amount is mostly distributed to them.

The Pool Manager accepts the price limit parameter, but it does not check the `amountOut` that the swapper receives. The severity is lowered due to the fact that even though the Pool Manager can be used directly and cannot detect this attack, swappers usually use a router which is able to verify exact `amountOut` value.

Attack scenario

The attackers might take the following steps in turn:

① The user submits a swap transaction with a specified price limit.
② The winner sees the transaction, front runs it and sets the swap fee to 100%.
③ The pool takes all tokens from user and makes swap with 0 amount without reaching the price limit.
④ The pool sends all tokens to the hook which distributes it to the liquidity providers and the winner.

Result of the attack: Strategist can steal the tokens that are swapped.

Recommendation

Consider adding a limit on maximum value of the `swapFee`.

References

1. SCSVS G4: Business logic

[ARB-d7f31318-L04] Inability to burn liquidity position

LOW PARTIALLY FIXED

Retest (2025-03-28)

The vulnerability has been fixed but the fix introduces a new issue. Whenever, the user wants to transfer the position it must unsubscribe first because otherwise the rewards will be accrued to the new owner.
CommitID: *5d3854a8b81ad9895d64cc0743e39c1ff88f8181*

Affected files

- RewardTracker.sol#L208

Description

The liquidity provider should use the position manager to add liquidity to pools that support a custom ERC20 token as the rent token. Later, the provider is able to manage their positions, including:

- increasing liquidity,
- decreasing liquidity,
- and burning the position.

However, when the provider burns their position, the transaction reverts. This happens because the `RewardTracker` accesses the owner of the position after it's burnt, leading to a revert.

```
206 _accrueRewards(
207     tokenId,
208     IERC721(address(positionManager)).ownerOf(tokenId),
209     liquidity,
210     pools[poolKey.toId()].getRewardsPerLiquidityInsideX128(
211         positionInfo.tickLower(),
212         positionInfo.tickUpper()
213     )
214 );
```

Vulnerable scenario

The following steps lead to the described result:

① The liquidity provider adds liquidity though the position manager.

② After some time, the provider burns the position through the position manager.
③ The position manager burns the position (resetting the owner) and modifies (clears) the liquidity.
④ The hook is notified about the liquidity modification and the `notifyModifyLiquidity` function is called.
⑤ The hooks gets the position owner using `ownerOf` function which reverts because the owner has been already reset.

Result: Inability to burn liquidity position.

Recommendation

Consider storing information about the position (e.g. owner) in the hook on subscription and clear it when the position is unsubscribed.

References

1. SCSVS G4: Business logic
2. SCSVS G8: Denial of service

[ARB-d7f31318-L05] Bypassing limits when changing strategies

LOW FIXED

Retest (2025-03-20)

The vulnerability has been fixed as recommended.
CommitID: *ccc3a9e3ac8f75337180a9bc66a1ca126fc7a133*

Affected files

- ArbiterAmAmmBaseHook.sol#L451

Description

The `changeStrategy` function is used by the current winner to change the address of strategy that is used to receive the current swap fee. The change should be executed only if the sender is current winner and the remaining rent is greater than zero.

However, the remaining rent variable is not updated before checking it's value. This allows the winner to update the strategy even if there is no remaining rent (assuming no new winner has been selected).

Vulnerable scenario

The following steps lead to the described result:

① The bidder becomes the winner for 10 blocks.
② Some swaps are executed in next 8 blocks but no swap is executed later.
③ The won 10 blocks have passed and user has no remaining rent but no new winner has been selected.
④ The winner is able to call `changeStrategy` function, because the `remainingRent` variable is still greater that 0 as it has not been updated in a swap, liquidity update or `overbid` function.

Result: Changing strategy address when the remaining rent is zero.

Recommendation

Call the `_payRentAndChangeStrategyIfNeeded` function before checking the value of remaining rent variable.

References

1. SCSVS G4: Business logic

[ARB-d7f31318-L06] Unnecessary cross for uninitialized ticks

LOW FIXED

Retest (2025-03-20)

The vulnerability has been fixed as recommended.
CommitID: *286b16f5d24f6fa0d8efa5a488330fa22c11e139*

Affected files

- PoolExtension.sol#L169-L181

Description

The `crossToActiveTick` function from `PoolExtension` library is used to track changes of the current tick in the pool and update the crossed ticks to correctly account for the rewards belonging to the tick ranges. Its logic is based on the PancakeSwap's library.

Whenever a tick is crossed its outside rewards are updated as the subtraction of the current

global rewards and the rewards stored in the tick struct. However, the function does not check whether the tick to be updated is already initialized. This leads to a situation when the outside reward is unnecessarily set for an uninitialized ticks.

The before-mentioned case happens when the current tick is moved away from the current word (e.g. by more than `256*tickSpacing` ticks). This is because the function `nextInitializedTickWith` returns the next initialized or uninitialized tick from the current word, up to 256 ticks away (multiplied by tick spacing) from the current tick.

Result: Unnecessary update of uninitialized ticks increasing the cost of swaps.

Recommendation

Call `crossTick` function only on initialized ticks. The information whether a tick is initialized is returned as the second returned value by `nextInitializedTickWithinOneWord` function.

References

1. SCSVS G4: Business logic

6. Recommendations

[ARB-d7f31318-R01] Consider using named mappings

INFO NOT IMPLEMENTED

Retest (2025-03-20)

The recommendation has not been implemented as recommended. There were inline docs added instead.

Description

Currently the key and id of mappings are neither named nor described in the comments for the following storage variables:

- `poolSlot0`
- `poolSlot1`
- `winners`
- `winnerStrategies`
- `deposits`
- `auctionFees`
- `pools`
- `positions`
- `accruedRewards`

Here is an example how the description of the keys and values can be moved to the variable's definition.

Before:

```
/// @notice Mapping of request IDs to their corresponding pending withdrawal
    amounts
/// @dev Key is the request ID, value is the amount of tokens to be withdrawn
mapping(uint256 => uint256) public pendingWithdrawalQueue;
```

After:

```
mapping(uint256 reqId => uint256 amount) public pendingWithdrawalQueue;
```

Recommendation

Use named mappings.

References

1. G11: Code clarity

[ARB-d7f31318-R02] Remove unnecessary code

INFO NOT IMPLEMENTED

Retest (2025-03-20)

The recommendation has been partially implemented as recommended.
The second recommendation has not been implemented as recommended, because the condition was kept. However, the calculation now does not revert„ but it returns a huge number that would probably overflow in the next instructions and revert anyway.

Description

There are some snippets in the source code that should not be present in the production-ready code.

Recommendation

- Remove the `return` keyword - ArbiterAmAmmBaseHook.sol#L556.
- Remove the second condition `rentEndBlock < uint32(block.number)` as such value would end with an underflow revert in L397 - ArbiterAmAmmBaseHook.sol#L354.

References

1. G11: Code clarity

[ARB-d7f31318-R03] Consider using newest Solidity version

INFO IMPLEMENTED

Retest (2025-03-28)

The recommendation has been implemented.

Description

In accordance with the best security practices, it is recommended to use the latest stable versions of major Solidity releases.

Very often, older versions contain bugs that have been discovered and fixed in newer versions. Moreover, it is worth remembering that the version should be clearly specified so that all tests and compilations are performed with the same version.

The current implementation uses:

```
pragma solidity ^0.8.26;
pragma solidity ^0.8.0;
```

Recommendation

Use the latest stable version of major Solidity release:

```
pragma solidity 0.8.28;
```

Note: If it is planned to deploy on multiple chains, stay aware that some of them don't support `PUSH0` opcode. If solc `>=0.8.20` is used, the `PUSH0` opcode will be present in the bytecode. In this situation, it is recommended to choose `0.8.19`.

References

1. SCSVS G1: Architecture, design and threat modeling
2. Floating pragma

[ARB-d7f31318-R04] Remove TODOs

INFO IMPLEMENTED

Retest (2025-03-20)

The recommendation has been implemented.

Description

Certain sections of the code contain lingering comments from the development team. These comments should be removed from the production codebase for improved clarity.

Recommendation

It is recommended to eliminate any remaining TODOs:

- ArbiterAmAmmBaseHook.sol#L22

References

1. G11: Code clarity

[ARB-d7f31318-R05] Use custom errors

INFO IMPLEMENTED

Retest (2025-03-20)

The recommendation has been implemented.

Description

Custom errors offer enhanced gas efficiency compared to traditional string messages and enable developers to provide detailed descriptions of errors utilizing NatSpec documentation.

Furthermore, starting from version 0.8.26 of Solidity, it is no longer necessary to negate the `require` expression in order to revert when utilizing custom errors, as they are now directly compatible with the `require` function.

Here are the places where custom errors can be introduced:

- RewardTracker.sol#L35-L38

Recommendation

Implement custom errors in place of string messages.
Custom errors have been supported by the `require` function since Solidity version 0.8.26.

```
require(
    balance[msg.sender] >= amount,
    InsufficientBalance(balance[msg.sender], amount)
);
```

References

1. SCSVS G11: Code Clarity

[ARB-d7f31318-R06] Fix comments

INFO IMPLEMENTED

Retest (2025-03-28)

The recommendation has been implemented as recommended.

Description

There are comments in the source code that do not reflect the logic implemented by the code.

- The comment in ArbiterAmAmmBaseHook.sol#L194-L198 states that in case of exact-in swap the fee currency is the one that user is selling, but in fact in both cases it's the currency that swapper is buying.

```
// Determine the specified currency. If amountSpecified < 0, the swap is exact
    -in so the feeCurrency should be the token the swapper is selling.
// If amountSpecified > 0, the swap is exact-out and it's the bought token.
bool exactOut = params.amountSpecified > 0;

bool isFeeCurrency0 = exactOut == params.zeroForOne;
```

- The comment in AuctionSlot0.sol#L12-L13 states that two variables are 8 bits and 16 bits respectively while in fact they are 1 bit and 23 bits.

```
[184..185): should change strategy (8 bits) - A boolean flag indicating
    whether a new strategy should be applied.
[185..208): winner fee part (16 bits) - The portion of the fee paid to
    the auction winner, expressed in basis points.
```

Recommendation

Change comments to properly reflect the code.

References

1. SCSVS G11: Code Clarity

[ARB-d7f31318-R07] Emit events for important state changes

INFO IMPLEMENTED

Retest (2025-03-20)

The recommendation has been implemented.

Description

The update of critical parameters should be tracked with events.

Recommendation

Emit an event in the following functions:

- ArbiterAmAmmBaseHook.setTransitionBlocks
- ArbiterAmAmmBaseHook.setMinRentBlocks
- ArbiterAmAmmBaseHook.setOverbidFactor
- ArbiterAmAmmBaseHook.setWinnerFeeSharePart
- ArbiterAmAmmBaseHook.setStrategyGasLimit
- ArbiterAmAmmBaseHook.setDefaultSwapFee
- ArbiterAmAmmBaseHook.setAuctionFee

References

1. SCSVS G1: Architecture, design and threat modeling
2. Principles and Best Practices to Design Solidity Events in Ethereum and EVM

7. Impact on risk classification

Risk classification is based on the one developed by OWASP [1], however it has been adapted to the immutable and transparent code nature of smart contracts. The Web3 ecosystem forgives much less mistakes than in the case of traditional applications, the servers of which can be covered by many layers of security.

Therefore, the classification is more strict and indicates higher priorities for paying attention to security.

OVERALL RISK SEVERITY				
Impact on risk	HIGH	CRITICAL	HIGH	MEDIUM
	MEDIUM	MEDIUM	MEDIUM	LOW
	LOW	LOW	LOW	INFO
		HIGH	MEDIUM	LOW
		Likelihood		

[1] OWASP Risk Rating methodology

8. Long-term best practices

8.1. Use automated tools to scan your code regularly

It's a good idea to incorporate automated tools (e.g. slither) into the code writing process. This will allow basic security issues to be detected and addressed at a very early stage.

8.2. Perform threat modeling

Before implementing or introducing changes to smart contracts, perform threat modeling and think with your team about what can go wrong. Set potential targets of the attacker and possible ways to achieve them, keep it in mind during implementation to prevent bad design decisions.

8.3. Use Smart Contract Security Verification Standard

Use proven standards to maintain a high level of security for your contracts. Treat individual categories as checklists to verify the security of individual components. Expand your unit tests with selected checks from the list to be sure when introducing changes that they did not affect the security of the project.

8.4. Discuss audit reports and learn from them

The best guarantee of security is the constant development of team knowledge. To use the audit as effectively as possible, make sure that everyone in the team understands the mistakes made. Consider whether the detected vulnerabilities may exist in other places, audits always have a limited time and the developers know the code best.

8.5. Monitor your and similar contracts

Use the tools available on the market to monitor key contracts (e.g. the ones where user's tokens are kept). If you have used code from another project, monitor their contracts as well and introduce procedures to capture information about detected vulnerabilities in their code.

Damian Rusinek

Smart Contracts Auditor

@drdr_zz

damian.rusinek@composable-security.com

Paweł Kuryłowicz

Smart Contracts Auditor

@wh01s7

pawel.kurylowicz@composable-security.com