# Security Review Report

January 3, 2026

# Athena V3 Core Level & Trader contracts

Lodelux

**Review window:** `2025-12-05` -> `2025-12-19`
**Security Researcher:** Lodelux
**Commit reviewed:** `403c7089000350ed98e0bc348d25a96909754dcd`
**Scope:**

- `contracts/TraderSpot/AthTrader.sol`
- `contracts/TraderSpot/AthQueue.sol`
- `contracts/TraderSpot/QueueInfo.sol`
- `contracts/TraderSpot/AthQueueExtension.sol`
- `contracts/TraderSpot/libraries/AthQueueLib.sol`
- `contracts/TraderSpot/PriceOracle.sol`
- `contracts/TraderSpot/QueueFactory.sol`
- `contracts/AthReferral/AthReferral.sol`
- `contracts/LpRewards/LpRewards.sol`
- `contracts/AthLevel/Zap.sol`
- `contracts/AthLevel/AthLevel.sol`

**Out Of Scope:** All the other files not listed in the above Scope section

**Disclaimer:** A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended. Nonetheless, I can assure you that I have done my absolute best and have treated this engagement with the utmost professionalism and respect for the client, users, and all stakeholders involved in this project.

**About Lodelux:** Lodelux is an independent Security Researcher with 3x Top 10 finishes in Audit Contests. He has found 50+ H/M bugs including 2 Solo findings.

## Contents

## Summary

The Security Review surfaced a moderate number of issues, some of them quite critical. Because of this, the Security Researcher advices the Athena Team to seek more actions towards solidifing the security of their smart contracts.

A direct suggestion is to implement a comprehensive test suite made of both unit tests and integration tests. This suite should verify and confirm that components and flows behave as the team intend them to, and should be quickly executable to verify that code changes do not break previously working functionalities. The actual implementation is up to the team as this would be classified as out of scope of this review, but a personal suggestion from the researcher is to use Foundry, which can be easily integrated in Hardhat and has actually been used by the Researcher to write Proof Of Concepts for this review.

### Methodology

The main focus of this review was high impact, high level issues/exploits. As such, the review was conducted mainly in a manual way and can be indexed in 3 main steps:

- Initial first pass to understand code *intentions*, map *main flows*, *roles* and *asset flows*
- Subsequent, multiple passes to understand what the code *actually* does and spot coding mistakes and inconsistencies with code intentions
- Olistic, deeper reviews of main flows with the goal to spot abstract and businness logic exploits
- Final, redundant scan with an AI auditor to increase confidence in issue coverage

Every issue discovered has been reviewed multiple times to assess its actual impact and validity, with the help of coded and runnable PoCs where deemed necessary.

### Identified Flows

Here is a non-comprehensive list of main flows that were identified and thoroughly analyzed. These are listed here to allow a better comprehension of this report and where in no way the only focus of the review nor all the identified flows:

- **ATHX_LP deposit/withdraw** - with and without Zap integration
- **Referral mechanics** - overall system design and incentive flows
- **LpRewards mechanics** - reward distribution and accrual logic
- **Queue/Trader lifecycle** - state transitions and component interactions
- **Trader swap execution** - with and without oracle price validation
- **Setter interactions** - global and contract-specific configuration side effects

### Risk Classification

To classify the risk of an issue, only its potential impact is considered, not its likelihood of happening. Although unrealistic or extremely improbable scenarios or pre-conditions are not considered and do not conform as base for a valid finding.

It was assumed that privileged roles (e.g. Admin or Owner) are trusted, act in the best interest of the protocol and can’t be compromised.

- **Critical:** Direct loss of funds or permanent protocol compromise with high likelihood.

- **High:** Loss of funds or severe disruption under realistic conditions.
- **Medium:** Meaningful risk; requires certain conditions or partial privilege.
- **Low:** Limited impact, edge-case, future-proofing or defense-in-depth improvement.
- **Informational:** Non-risk or best-practice issue; improves clarity/maintainability.

### Findings Snapshot

| Severity | Count |
|---|---:|
| Critical | 2 |
| High | 3 |
| Medium | 14 |
| Low/Info | 27 |
| Total | 46 |

## Detailed Findings

### C-01 - Investor can avoid losses by using `withdraw()` instead of `claimProfit()` rendering queue insolvent

**Severity:** Critical
**Code references:** `AthQueueExtension::withdrawInvestment`

#### Description

```
function withdrawInvestment() external nonReentrant {
    if (!dependenciesSet) revert DepsNotSet();
    if (
        currentState != IAthQueue.QueueState.standby &&
        currentState != IAthQueue.QueueState.init
    ) revert CannotWithdrawInThisState();

    address investorAddr = msg.sender;
    IAthQueue.InvestorData storage investor = investorData[investorAddr];

@>  uint256 amount = investor.standByAmount;
    if (amount == 0) revert InsufficientStandbyAmount();

    investor.standByAmount = 0;
    investor.totalInvest -= uint128(amount);
    investor.entryCycleId = 0;
    investor.costBasis = 0;
    investor.lifetimeFeePaid = 0;
    totalStandyAmount -= amount;

    if (investor.totalInvest == 0 && isInvestor[investorAddr]) {
        isInvestor[investorAddr] = false;
        totalInvestors--;
    }

    address recipient = investorAddr;
    if (IUserRegistry(userRegistry).isBlacklisted(investorAddr)) {
        recipient = _athLevel.getErrorWallet();
        if (recipient == address(0)) revert ErrorWalletNotSet();
    }

@>  IERC20(participationToken).safeTransfer(recipient, amount);
    emit WithdrawInvestment(investorAddr, amount);
}
```

When a user calls `withdraw`, he gets back all of his `standByAmount`. This is effectively the user's original deposit accounting, and it is **not** updated to reflect the outcome of the latest trading cycle.

Concretely, when a cycle starts the contract aggregates liquidity and transfers `totalStandyAmount` to the trader, but it does not "settle" each investor's position in storage. At the end of a cycle, the trader returns funds and the queue switches back to `standby`, but per-user PnL is only realized when the user calls `claimProfit()`: `claimProfit()` calls `_moveToTrading()` (which moves `standByAmount` into `tradingAmount` once the cycle has advanced) and then computes the user's final redeemable amount via `AthQueueLib.calculateFinalAmount01(cycleHistory, investor.costBasis, investor.entryCycleId)`.

`withdrawInvestment()` bypasses this entire settlement path: it never calls `_moveToTrading()` and never applies `cycleHistory` to the user's balance, so the user can always redeem their pre-trade `standByAmount` after an unprofitable cycle.

**Impact**

This allows a user to avoid trading losses by simply calling `withdraw` instead of `claimProfit` at the end of an unprofitable trading cycle, the "extra" funds are effectively stolen from other investors and the queue will be insolvent.

**Recommendation**

Redesign withdraw/claim mechanincs to correctly reflects users' assets. As the fix is quite subjective, I can't provide a direct, actionable code fix.

**Notes** fixed in `38f791e`

## C-02 - Oracle price truncation for sub-$1 tokens bypasses slippage validation

**Severity:** Critical
**Code references:**

- `PriceOracle::_validateWithPyth`
- `PriceOracle::_convertPythPrice`
- `AthTrader::swap`

### Description

Pyth price feeds are structured as:

```
struct Price {
    int64 price;       // The price of the asset
    int32 expo;        // The exponent to scale the price by (10^expo)
    uint64 conf;      // The confidence interval around the price
    uint32 publishTime; // The timestamp when the price was published
}
```

and as it can be seen in their official docs https://api-reference.pyth.network/price-feeds/evm/getPriceNoOlderThan, scaling the given `price` by `expo` returns the actual price in $, with no added precision.

The oracle attempts to convert Pyth prices into a `uint256` USD price via `_convertPythPrice`, but for negative exponents it performs an integer division:

```
function _convertPythPrice(PythStructs.Price memory price) internal pure
↪   returns (uint256) {
    require(price.price > 0, "Invalid price");

    int64 priceValue = price.price;
    int32 expo = price.expo;

    if (expo >= 0) {
        return uint256(uint64(priceValue)) * (10 ** uint32(expo));
    } else {
@>      return uint256(uint64(priceValue)) / (10 ** uint32(-expo));
    }
}
```

This effectively truncates all fractional prices. As a result, any token priced below `$1` (e.g., `$0.50`) will be converted into `0`, which then propagates into the swap validation math:

```
function _validateWithPyth(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 expectedOut
) internal returns (bool valid, string memory reason) {
    // ...
    uint256 priceInUSD = _convertPythPrice(priceIn);
    uint256 priceOutUSD = _convertPythPrice(priceOut);

    uint256 decimalsIn = IATHERC20(tokenIn).decimals();
    uint256 decimalsOut = IATHERC20(tokenOut).decimals();

@>  uint256 valueInUSD = (amountIn * priceInUSD) / (10 ** decimalsIn);
    uint256 oracleExpectedOut = (valueInUSD * (10 ** decimalsOut)) /
    ↪   priceOutUSD;
    uint256 minAcceptable = (oracleExpectedOut * (10000 - maxSlippage)) /
    ↪   10000;

    if (expectedOut < minAcceptable) {
        // ...
    }
    // ...
}
```

If `priceInUSD == 0`, then `valueInUSD == 0`, `oracleExpectedOut == 0`, and thus `minAcceptable == 0`. This causes the slippage check to accept any `expectedOut`, including `0`.

Since `AthTrader::swap` forwards the trader-controlled `amountOutMin_` to `PriceOracle.validateSwap(...)` as `expectedOut`, a malicious trader can set `amountOutMin_ == 0` and bypass oracle protections for these low-priced tokens:

```
function swap(
    address router_,
    address[] memory route_,
    uint256 amountIn_,
    uint256 amountOutMin_,
    uint256 deadline_,
    bytes calldata priceUpdateData
) external payable onlyTrader nonReentrant {
    // ...
@>  (bool valid, string memory reason) =
↪   IPriceOracle(oracleAddress).validateSwap{value: oracleFee}(
        tokenIn,
        tokenOut,
        amountIn_,
        amountOutMin_,
        priceUpdateData
    );
    require(valid, reason);
    // ...
}
```

This can be easily exploited by a malicious trader by basically performing a same-tx sandwich attack:

- trader flashloans huge amount of tokenA
- trader skews tokenA/tokenB pool by swapping this large amount directly
- trader swaps all of the queue's tokenA balance for very few tokenB
- trader swaps his tokenB for tokenA
- trader repays flashloan, pocketing profit stolen from the queue

**Impact**

For any whitelisted token with a price below `$1`, the oracle slippage check can become a no-op. A malicious trader can then execute swaps at arbitrarily bad rates (after manipulating the pool) and extract funds from the queue/trader via LP share + unwind, resulting in loss of user funds.

**Recommendation**

switch price conversion to `PythUtils.convertToUint` to correctly convert oracle prices to a safe precision, e.g. 18 decimals.

**Notes** fixed in `ffc5c69`

## H-01 - `safeApprove()` non-zero->non-zero can be weaponized to DoS future zaps

**Severity:** High

**Code references:**

- `Zap::zapAndDeposit`
- `AthLevel::depositLPFor`
- `AthLevel::_afterLPReceived`

### Description

`Zap::zapAndDeposit()` grants the Level contract an allowance for LP using `SafeERC20.safeApprove()` immediately before calling `level.depositLPFor(...)` inside a `try/catch`:

```
function zapAndDeposit(uint256 usdtAmount, address referrer) external
↪   nonReentrant {
    // ...
    (,, uint256 liquidity) = router.addLiquidity(
        address(USDT),
        address(ATHX),
        usdtBal,
        athxBal,
        minAmountA,
        minAmountB,
        address(this),
        block.timestamp + 1200
    );
    require(liquidity > 0, "no LP minted");

    // Cleanup: reset router approvals
    USDT.safeApprove(address(router), 0);
    ATHX.safeApprove(address(router), 0);

@>  LP.safeApprove(address(level), liquidity);

@>  try level.depositLPFor(msg.sender, liquidity, usdtAmount, referrer) {
        emit ZapExecuted(msg.sender, usdtAmount, liquidity, usdtAmount);
    } catch Error(string memory reason) {
        LP.safeTransfer(msg.sender, liquidity);
        emit ZapFailed(msg.sender, liquidity, reason);
    } catch (bytes memory) {
        LP.safeTransfer(msg.sender, liquidity);
        emit ZapFailed(msg.sender, liquidity, "unknown error");
    }

    _returnDust(msg.sender);
}
```

However, `SafeERC20.safeApprove()` reverts when attempting to change an existing allowance from a non-zero value to another non-zero value.

If `depositLPFor()` reverts for any reason (and the revert is caught), the approval is **not consumed** (the transferFrom is reverted inside the failed external call), and it is **not reset to zero**. This leaves a non-zero LP allowance from Zap -> `AthLevel`.

From that point on, the next `zapAndDeposit()` call (from any user) will revert at

`LP.safeApprove(address(level), liquidity)`, permanently breaking the zap flow for everyone.

A malicious user can intentionally trigger this state by ensuring `depositLPFor()` reverts while still allowing the zap to mint LP. For example, if the user is in the "unlocking" state, `AthLevel::_afterLPReceived()` reverts:

```
function depositLPFor(address user_, uint256 lpAmount, uint256 usdCredited,
↪   address referrer) external nonReentrant {
    // ...
    lpToken.safeTransferFrom(msg.sender, address(this), lpAmount);
    // ...
    _afterLPReceived(user_, lpAmount, usdCredited);
}

function _afterLPReceived(address user_, uint256 lpAmount, uint256
↪   usdCredited) private {
    User storage u = users[user_];
@>  require(u.unlockTime == 0, "Cannot deposit while unlocking");
    // ...
    u.activeLP += lpAmount;
    u.activeUSD += usdCredited;
    // ...
}
```

**Impact**

Any user can permanently DoS `Zap::zapAndDeposit()` for all users, requiring a Zap redeploy to restore functionality.

**Recommendation**

use `safeIncreaseAllowance` as `safeApprove` is actually deprecated

**Notes** fixed in `e7ebbe7`

## H-02 - `invest()` can revert due to duplicate record, blocking reinvest for the user indefinetly

**Severity:** High
**Code references:**

- `AthQueueExtension::invest`
- `AthQueueExtension::withdrawInvestment`
- `UserRegistry::recordContract`

### Description

`AthQueueExtension::invest()` attempts to register the user inside `UserRegistry` when the queue considers them a "new investor" (`!isInvestor[investorAddr]`):

```
function invest(uint256 amount_, address _referrer) external nonReentrant {
    // ...
    IAthQueue.InvestorData storage investor = investorData[investorAddr];
    if (
        investor.tradingAmount != 0 ||
        investor.standByAmount != 0 ||
        investor.entryCycleId != 0
    ) revert MustExitCompletely();

    IERC20(participationToken).safeTransferFrom(investorAddr,
 ↪  address(this), amount_);
    if (!isInvestor[investorAddr]) {
        investorsAddress.push(investorAddr);
        isInvestor[investorAddr] = true;
        totalInvestors++;
@>      IUserRegistry(userRegistry).recordContract(
            investorAddr,
            TYPE_QUEUE,
            uint64(uint160(address(this))),
            ROLE_INVESTOR
        );
    }
    investor.standByAmount = uint128(amount_);
    investor.totalInvest += uint128(amount_);
    // ...
}
```

`UserRegistry::recordContract()` is not idempotent and reverts on duplicates:

```
function recordContract(address user, uint8 typeId, uint64 contractId,
 ↪  uint8 role)
```

```
    external
    onlyAuthorizedForType(typeId)
    whenNotPaused
    dependenciesAreSet
{
    require(user != address(0), "Invalid user");
    require(typeId > 0, "Invalid typeId");
@>  require(!_hasRecord[user][typeId][contractId][role], "Already
 ↪  recorded");
    // ...
}
```

The queue also “unregisters” an investor from its local bookkeeping when they withdraw fully and `investor.totalInvest == 0`:

```
function withdrawInvestment() external nonReentrant {
    // ...
    investor.totalInvest -= uint128(amount);
    totalStandyAmount -= amount;

    if (investor.totalInvest == 0 && isInvestor[investorAddr]) {
@>      isInvestor[investorAddr] = false;
        totalInvestors--;
    }
    // ...
}
```

However it DOES NOT unrecord him from `userRegistry`, As a result, a user that:

1. invests once, and later
2. exits completely such that `isInvestor` is flipped back to `false`, and then
3. tries to invest again,
   will hit `recordContract()` again and revert with `"Already recorded"`.

This blocks reinvestment indefinitely for that user in that queue, because the `UserRegistry` entry persists and there is no corresponding “unrecord”/cleanup on exit.

**Impact**

Users can be permanently prevented from investing again into the same queue after exiting completely, causing a severe disruption of normal protocol usage.

**Recommendation**

Ensure user registry recording is compatible with reinvestment, by recording only if needed.

**Notes** fixed in 88bdb53

### H-03 - Slippage protection uses `getAmountsOut` (skewed), making mins unreliable

**Severity:** High
**Code references:** `Zap::zapAndDeposit`

#### Description

`Zap::zapAndDeposit()` attempts to compute slippage protection for the USDT->ATHX swap by calling `router.getAmountsOut()` and applying `slippageBps`:

```
function zapAndDeposit(uint256 usdtAmount, address referrer) external
↪   nonReentrant {
    // ...
    uint256 half = usdtAmount / 2;
    address[] memory path = new address[](2);
    path[0] = address(USDT);
    path[1] = address(ATHX);

@>  uint256[] memory amountsOut = router.getAmountsOut(half, path);
@>  uint256 minAthxOut = (amountsOut[1] * (10000 - slippageBps)) / 10000;

    USDT.safeApprove(address(router), 0);
    USDT.safeApprove(address(router), half);
    router.swapExactTokensForTokens(
        half,
        minAthxOut,
        path,
        address(this),
        block.timestamp + 1200
    );
    // ...
}
```

This does **not** provide meaningful slippage protection for the user because `getAmountsOut()` reads the pool reserves **at execution time**, which could be skewed and not reflect an honest ratio. Even though the ATHX token implements a MEV-bot protecions, this does not solve this issue as the ratio of the pool could be skewed simply by a big enough trade.

#### Impact

The user has no real way to set a minimum output for his zap, resulting in plausible scenarios where he receieves less LPs than he normally should.

**Recommendation**

Do not derive `amountOutMin` from `getAmountsOut()` inside the same transaction.

- Accept a user-supplied `minAthxOut` (computed off-chain by the frontend) and pass it directly to the swap.
- Or enforce an oracle/TWAP-based bound (e.g., compare the implied swap price vs a Pyth/TWAP reference and revert if deviating beyond `slippageBps`).

**Notes** fixed in `a2d2679`

### M-01 - `lpDeposit` return value can override admin-given level

**Severity:** Medium
**Code references:**

- `AthLevel::setUserLevel`
- `AthLevel::depositLP`
- `AthLevel::_afterLPReceived`

#### Description

`AthLevel::setUserLevel()` allows a `levelManager` to set a user's `u.level` directly:

```
function setUserLevel(address user_, uint256 level_) external
 ↪  onlyLevelManager {
    require(level_ <= maxLevel, "Level exceeds max");
    User storage u = users[user_];
@>  u.level = uint8(level_);
    // ...
    emit LevelChanged(user_, level_);
}
```

However, any subsequent LP deposit re-computes the user's level from `u.activeUSD` and overwrites `u.level` if the computed value differs:

```
function depositLP(uint256 lpAmount, address referrer) external
 ↪  nonReentrant {
    // ...
@>  _afterLPReceived(msg.sender, lpAmount, 0);
}

function _afterLPReceived(address user_, uint256 lpAmount, uint256
 ↪  usdCredited) private {
    User storage u = users[user_];
    // ...
    uint256 newLevel = _calcLevelFromUSD(u.activeUSD);
    if (newLevel != u.level) {
@>      u.level = uint8(newLevel);
        emit LevelChanged(user_, newLevel);
    }
    // ...
}
```

Because `setUserLevel()` does not adjust `u.activeUSD` to match the manually set `u.level`, the next `_afterLPReceived()` call will "snap" the user back to the computed level. This makes `setUserLevel()` non-durable and can lead to confusing behavior (e.g., a manually assigned level being silently reverted on a later deposit).

**Impact**

Level managers cannot reliably grant (or enforce) a manual level since it can be overwritten by the next deposit, leading to inconsistent user privileges and a support burden to explain or restore expected levels.

**Recommendation**

Redesign manual level setters, consider modifying `u.activeUSD` rather than level directly.

**Notes** fixed in `ee4ae8b`

### M-02 - Zap add-liquidity mins (99%) + split swap can frequently revert

**Severity:** Medium
**Code references:** `Zap::zapAndDeposit`

**Description**

`Zap::zapAndDeposit()` swaps exactly half of the input USDT into ATHX and then attempts to add liquidity with `amountAMin`/`amountBMin` set to ~99% of the post-swap balances:

```
function zapAndDeposit(uint256 usdtAmount, address referrer) external
 ↪  nonReentrant {
    // ...
    uint256 half = usdtAmount / 2;
    // swapExactTokensForTokens(half, ...)

    uint256 usdtBal = USDT.balanceOf(address(this));
    uint256 athxBal = ATHX.balanceOf(address(this));

@>  uint256 minAmountA = (usdtBal * (10000 - liquiditySlippageBps)) /
 ↪  10000;
    uint256 minAmountB = (athxBal * (10000 - liquiditySlippageBps)) /
     ↪  10000;

    (,, uint256 liquidity) = router.addLiquidity(
        address(USDT),
        address(ATHX),
        usdtBal,
        athxBal,
        minAmountA,
        minAmountB,
        address(this),
        block.timestamp + 1200
    );
    // ...
}
```

On UniswapV2-style routers, `addLiquidity()` uses the *pool price* to determine the "optimal" amounts, and will leave dust of the over-supplied token. Since the zap always swaps "half" (instead of swapping to match the pool ratio), it is common for one side to be over-supplied under normal market conditions. In those cases, requiring ~99% of both token balances to be consumed makes `addLiquidity()` revert (e.g., `INSUFFICIENT_A_AMOUNT` / `INSUFFICIENT_B_AMOUNT`) even for honest users.

Consider this reverting scenario without any MEV/manipulation and with the maximum allowed slippage of 10%:

1. the pool is at a 1:1 ratio (e.g., `reserveUSDT = 1000`, `reserveATHX = 1000`).
2. Configure `liquiditySlippageBps = 1000` (10%), meaning the contract requires `amountMin = 90%` of the post-swap balances.
3. Call `zapAndDeposit(1000 USDT)`:
   - The zap swaps `half = 500 USDT` for ATHX (UniswapV2 pricing with 0.3% fee), receiving `~332.66 ATHX` due to price impact.
   - The zap now holds `usdtBal ~= 500` and `athxBal ~= 332.66`.
4. When calling `addLiquidity(usdtBal, athxBal, minAmountA, minAmountB, ...)`:
   - The pool ratio after the swap is ~`reserveUSDT = 1500`, `reserveATHX = 667.34`.
   - The router computes the optimal ATHX needed for `500 USDT` as `amountBOptimal = quote(500, 1500, 667.34) ~= 222.44 ATHX`.
   - But the contract sets `minAmountB = 90% * 332.66 ~= 299.4 ATHX`, so `amountBOptimal < minAmountB` and `addLiquidity()` reverts with `INSUFFICIENT_B_AMOUNT`.

This shows that even with a loose 10% setting (not only 99%), the "swap half + require ~all balances added" approach is inherently incompatible with UniswapV2-style `addLiquidity()` behavior.

**Impact**

The zap flow becomes unreliable: legitimate zaps can revert frequently and since zapping is the only way for a user to get a level, this breaks a core functionality of the protocol.

**Recommendation**

Align the swap amount with the pool ratio and set mins relative to the *expected* amounts used:

- Compute the required swap to target the pool ratio (e.g., using reserves/quote logic), then add liquidity with `amountAMin/amountBMin` derived from those expected amounts and `liquiditySlippageBps`.
- Alternatively, accept user-supplied `amountAMin/amountBMin` from the frontend and avoid enforcing "consume ~all balances" semantics in-contract.

**Notes** fixed in `513048c`

### M-03 - Given level is inflated w.r. to actual deposited USDT

**Severity:** Medium
**Code references:**

- `Zap::zapAndDeposit`
- `Zap::_returnDust`
- `AthLevel::_afterLPReceived`

#### Description

`Zap::zapAndDeposit()` credits the user with `usdCredited = usdtAmount` when calling `AthLevel.depositLPFor(...)`, regardless of how much USDT was actually used by `addLiquidity()`:

```
function zapAndDeposit(uint256 usdtAmount, address referrer) external
↪   nonReentrant {
    // ...
    (,, uint256 liquidity) = router.addLiquidity(
        address(USDT),
        address(ATHX),
        usdtBal,
        athxBal,
        minAmountA,
        minAmountB,
        address(this),
        block.timestamp + 1200
    );
    require(liquidity > 0, "no LP minted");

@>  try level.depositLPFor(msg.sender, liquidity, usdtAmount, referrer) {
        emit ZapExecuted(msg.sender, usdtAmount, liquidity, usdtAmount);
    } catch {
        // ...
    }

    _returnDust(msg.sender);
}
```

But `addLiquidity()` may consume materially less than `usdtBal` (leaving USDT dust) whenever the inputs are not already in the exact pool ratio. The zap then returns any leftover USDT/ATHX to the user via `_returnDust()`, meaning the user can be credited for USD that was **not actually deposited into the level system**.

This is especially problematic because `AthLevel` derives `u.level` from `u.activeUSD`:

```
function _afterLPReceived(address user_, uint256 lpAmount, uint256
 ↪   usdCredited) private {
    User storage u = users[user_];
    // ...
@>  u.activeUSD += usdCredited;
    uint256 newLevel = _calcLevelFromUSD(u.activeUSD);
    // ...
}
```

**Impact**

Users can obtain a higher `AthLevel` than intended relative to the actual USDT value deposited (and kept) by the system, potentially unlocking benefits/requirements that are meant to be gated by real deposited value.

**Recommendation**

Pass the *actual* USDT amount used to mint LP as `usdCredited`:

Capture `amountA/amountB` from `addLiquidity()` and pass `amountA` (USDT used) to `depositLPFor(...)`, not the original `usdtAmount`.

**Notes** fixed in `0c8064a`

### M-04 - Malicious user can force `depositLPFor` call from victim to fail; user gets LP but no level

**Severity:** Medium
**Code references:**

- `Zap::zapAndDeposit`
- `AthLevel::depositLPFor`

**Description**

`Zap::zapAndDeposit()` performs swaps + adds liquidity, then wraps the final `level.depositLPFor(...)` step in a `try/catch`. If the level deposit fails, the zap does not revert; instead it returns the LP tokens to the caller:

```
function zapAndDeposit(uint256 usdtAmount, address referrer) external
↪   nonReentrant {
    // ...
    LP.safeApprove(address(level), liquidity);

    try level.depositLPFor(msg.sender, liquidity, usdtAmount, referrer) {
        emit ZapExecuted(msg.sender, usdtAmount, liquidity, usdtAmount);
    } catch Error(string memory reason) {
@>      LP.safeTransfer(msg.sender, liquidity);
        emit ZapFailed(msg.sender, liquidity, reason);
    } catch (bytes memory) {
        LP.safeTransfer(msg.sender, liquidity);
        emit ZapFailed(msg.sender, liquidity, "unknown error");
    }

    _returnDust(msg.sender);
}
```

There could be multiple ways for `depositLPFor()` to revert (including failures in referral recording), which means a user can end up holding LP while not receiving any level credit:

```
function depositLPFor(address user_, uint256 lpAmount, uint256 usdCredited,
↪   address referrer)
    external
    nonReentrant
{
    // ...
    if (referrer != address(0) && address(referral) != address(0)) {
@>      referral.recordReferral(user_, referrer);
    }
    _afterLPReceived(user_, lpAmount, usdCredited);
}
```

If an attacker can induce conditions that make `recordReferral(...)` revert for a victim (see M-05), the victim's zap can be forced into the "LP returned, no level credited" path. In practice, the victim can no longer obtain a level increase via zap unless they change parameters (e.g., omit `referrer`) or receive manual intervention, since `AthLevel.depositLP(...)` credits `usdCredited = 0`.

**Impact**

Victims can be griefed into paying gas for zaps that do not increase their level, degrading the intended UX and potentially blocking access to level-gated features until support/admin intervention.

**Recommendation**

Make the zap outcome atomic: avoid partial success by reverting the entire zap if `depositLPFor()` fails.

**Notes** fixed in `d2bad1d`

## M-05 - Malicious user can block victim from being able to ever add referrals

**Severity:** Medium
**Code references:**

- `AthReferral::_addUserReferrer`
- `AthReferral::recordReferral`

### Description

`AthReferral::_addUserReferrer()` enforces that a user can only set their referrer if they currently have *no referees*:

```
function _addUserReferrer(address _referrer, address _user) private {
    // ...
    if (referrerOf[_user] == address(0)) {
@>      require(refereesOf[_user].length == 0, "User already has referees,
 ↪  contact support");
        referrerOf[_user] = _referrer;
        refereesOf[_referrer].push(_user);
    }
}
```

This check is problematic because `refereesOf[_user]` can be populated *by other users*, without `_user`'s consent: any new address can set `_user` as their referrer through an authorized integration (e.g., deposit with `_user` as `referrer`). After that single action, `_user` will have at least one referee, and any later attempt to set `_user`'s own referrer will revert with `"User already has referees, contact support"`.

Because `recordReferral(...)` calls `_addUserReferrer(...)` and does not catch reverts, this also creates a "hard" grief vector: once a victim has referees, integrations that attempt to set a referrer for the victim will revert.

### Impact

An attacker can permanently prevent a victim from setting their own referrer (and therefore from participating in the intended referral tree as a referred user), causing long-term loss of referral benefits and creating avoidable transaction failures for normal flows that record referrals.

### Recommendation

Remove or soften the "no referees" restriction:

- Allow users to have both a referrer and referees (enforce only circularity checks).

- If the intent is to prevent late referrer assignment, replace the `require` with a non-reverting no-op (skip setting the referrer) so third-party actions cannot permanently brick user flows.

**Notes** fixed in `5ee8d2b`

### M-06 - Total loss (0 returned) can brick queue/contracts

**Severity:** Medium
**Code references:** `AthQueue::concludeTradingContract`

#### Description

`AthQueue::concludeTradingContract()` reverts if `returnFromCycle_ == 0`:

```
function concludeTradingContract(uint256 returnFromCycle_) external
↪   nonReentrant {
    // ...
@>  if (returnFromCycle_ == 0) revert NoFundsReceived();
    // ...
}
```

In a total-loss scenario, the trader may return `0`, causing `concludeTradingContract()` to revert. Because this is the only transition out of the `trading` state (and is only callable by `traderContract`), the queue can become stuck in `trading`, making subsequent trading cycles impossible to start.

#### Impact

A total-loss cycle can permanently DoS the queue by making `concludeTradingContract()` impossible to execute, preventing all future trading cycles and trapping the queue lifecycle in the `trading` state.

#### Recommendation

It is assumed that a total loss scenario should be valid, as there is a note in code saying just that. So it is recommended to:

- Allow `returnFromCycle_ == 0` to conclude the cycle and transition back to `standby`.
- Ensure downstream accounting (e.g., checkpoint and final amount calculations) is compatible with a 0-value outcome without reverting.

**Notes** fixed in `0632ba3`

### M-07 - Missing `userRegistry.recordContract` / unrecord for added/removed traders

**Severity:** Medium
**Code references:**

- `AthQueue::addTrader`
- `QueueFactory::_deployQueue`

**Description**

On initial deployment, the factory registers both deployer and all `authorizedTraders_` in `UserRegistry` with `ROLE_MANAGER`:

```
function _deployQueue(/* ... */) internal returns (address) {
    // ...
    if (userRegistry != address(0) && authorizedTraders_.length > 0) {
        for (uint256 i = 0; i < authorizedTraders_.length; i++) {
            address trader = authorizedTraders_[i];
            // ...
@>          try IUserRegistry(userRegistry).recordContract(trader,
↪   TYPE_QUEUE, uint64(uint160(clone)), ROLE_MANAGER) {
                // ...
            } catch {
                // ...
            }
        }
    }
    // ...
}
```

However, the post-deployment trader management functions on the queue only update `authorizedTraders` and emit events, without updating `UserRegistry`:

```
function addTrader(address traderAddress) external {
    _onlyOwnerOrDeployer();
    // ...
@>  authorizedTraders[traderAddress] = true;
    emit TraderAdded(traderAddress);
}

function removeTrader(address traderAddress) external {
    _onlyOwnerOrDeployer();
    // ...
@>  authorizedTraders[traderAddress] = false;
```

```
    emit TraderRemoved(traderAddress);
}
```

This causes the “source of truth” for traders to diverge:

- newly-added traders are authorized on-chain but are missing from `UserRegistry` (so any UI/indexer relying on `UserRegistry` won’t reflect the actual authorization state),
- removed traders can remain registered as `ROLE_MANAGER` even after losing authorization.

**Impact**
Trader authorization can become inconsistent across protocol components and off-chain infrastructure, leading to incorrect UI displays and potential operational issues if any downstream system relies on `UserRegistry` for determining current authorized traders/managers.

**Recommendation**
Keep trader authorization and `UserRegistry` synchronized:

- On `addTrader`, call `recordContract(trader, TYPE_QUEUE, queueId, ROLE_MANAGER)` (preferably idempotent).
- On `removeTrader`, implement a compatible “unrecord”/revocation flow (e.g., `UserRegistry` removal support, or role downgrade via `updateRole`, or alternatively update the frontend to treat `UserRegistry` as historical and always query `AthQueue.authorizedTraders` as the authoritative source).

**Notes** fixed in `925f605`

### M-08 - `claiming` does not decrement `totalInvestors` after full withdrawal, locking new investors out of queue

**Severity:** Medium

**Code references:**

- `IAthQueue::InvestorData`
- `AthQueue::invest`
- `AthQueueExtension::_processClaim`
- `AthQueueExtension::withdrawInvestment`

#### Description

The queue enforces `maxInvestors` using the `totalInvestors` counter:

```
function invest(uint256, address) external {
    uint8 max = maxInvestors > 0 ? maxInvestors :
     ↪  IQueueInfo(queueInfo).maxInvestorsPerQueue();
@>  require(totalInvestors < max, "Max investors reached");
    _delegate();
}
```

However, a user who fully exits via `claimProfit()` is not removed from the `totalInvestors` count. In `_processClaim()`, the user's position is settled and internal accounting is reset, but `isInvestor` / `totalInvestors` are never updated:

```
function _processClaim(address investorAddr, bool forceErrorWallet) private
 ↪  {
    // ...
    investor.claimedAmount += uint128(netReward);
    investor.tradingAmount = 0;
    investor.costBasis = 0;
    investor.entryCycleId = 0;
    investor.lifetimeFeePaid = 0;
    // ...
    IERC20(participationToken).safeTransfer(recipient, netReward);
    emit ClaimProfit(investorAddr, netReward);
}
```

Additionally, note that investors are only removed when `investor.totalInvest == 0`, and `totalInvest` is not decreased on `claimProfit()` (it is treated as a lifetime metric). Instead, `totalInvest` is only decreased on `withdrawInvestment()` (standby withdrawals):

```
function withdrawInvestment() external nonReentrant {
    // ...
    investor.standByAmount = 0;
    investor.totalInvest -= uint128(amount);
    // ...
@>  if (investor.totalInvest == 0 && isInvestor[investorAddr]) {
        isInvestor[investorAddr] = false;
        totalInvestors--;
    }
    // ...
}
```

As a result, `totalInvestors` can only ever decrease for users who withdraw from standby (pre-trade). Any investor that has ever claimed at least once will typically keep a non-zero `totalInvest`, and will not be removed from `totalInvestors` even after fully exiting via `claimProfit()`.

**Impact**

Queues can become permanently "full": once `totalInvestors` reaches `maxInvestors`, new investors are blocked from joining even if many (or all) previous investors have fully exited via claiming.

**Recommendation**

When a claim results in the user having no remaining position (`standByAmount == 0` and `tradingAmount == 0`), remove them from the investor set:

- set `isInvestor[investorAddr] = false` and decrement `totalInvestors`,
- remove `totalInvest == 0` as criteria to remove investor

**Notes** fixed in d5b3d36

### M-09 - `QueueFactory` call to `QueueInfo.setFactory` will always revert, bricking deployment

**Severity:** Medium
**Code references:**

- `QueueFactory::setQueueInfo`
- `QueueInfo::setFactory`

#### Description

`QueueFactory::setQueueInfo()` attempts to "auto-authorize" itself in `QueueInfo` by calling `QueueInfo.setFactory(address(this), true)`:

```
function setQueueInfo(address queueInfo_) external onlyOwner {
    // ...
    queueInfoAddr = queueInfo_;
@>  IQueueInfo(queueInfo_).setFactory(address(this), true);
}
```

But `QueueInfo::setFactory()` is `onlyOwner`:

```
function setFactory(address factory, bool authorized) external onlyOwner {
    // ...
    authorizedFactories[factory] = authorized;
}
```

Therefore, unless `QueueInfo` is owned by the `QueueFactory` contract itself (which is not the deployment pattern), the call will revert, preventing `setQueueInfo()` from ever succeeding.

Since `queueInfoAddr` is a required dependency for deployments, this can brick queue creation at the deployment/bootstrap phase.

#### Impact

Protocol deployment can be blocked at bootstrap time (queue creation becomes impossible) unless the system is redeployed or manual ownership/authorization steps are changed.

#### Recommendation

Remove this cross-contract `onlyOwner` call from `QueueFactory::setQueueInfo()`:

- Require the `QueueInfo` owner to authorize the factory separately by calling `QueueInfo.setFactory(factory, true)`.

- If you want to keep auto-authorization, change the access control to a factory-appropriate method (e.g., `onlyOwnerOrAdmin`) and/or wrap the call in `try/catch` and surface a clear deployment checklist.

**Notes** fixed in `16c8975`

### M-10 - Anti-pattern: `tx.origin`-style assumptions / contract caller restrictions

**Severity:** Medium
**Code references:** `QueueFactory::createQueue`

#### Description

`QueueFactory::createQueue()` enforces EOA-only access patterns:

```
function createQueue(/* ... */) external payable dependenciesAreSet returns
↪   (address) {
    // ...
@>  require(tx.origin == msg.sender, "Deployer must be EOA");

    for (uint i = 0; i < authorizedTraders_.length; i++) {
        address trader = authorizedTraders_[i];
        uint256 size;
        assembly { size := extcodesize(trader) }
@>      require(size == 0, "Traders must be EOA");
    }
    // ...
}
```

These restrictions are a common anti-pattern:

- They block legitimate users who deploy via multisigs/smart wallets (e.g., Gnosis Safe), and are incompatible with account abstraction patterns.
- They provide a false sense of security: `tx.origin` checks do not meaningfully prevent malicious behavior (they only restrict composability).

#### Impact

Legitimate governance and operational setups (multisig/timelock-based deployments) can be prevented from creating queues, forcing the protocol to rely on EOAs for sensitive operations and reducing compatibility with common ecosystem tooling.

#### Recommendation

Remove the `tx.origin`/EOA-only restrictions

**Notes** fixed in b93b542

### M-11 - Catch block allows for an inconsistent state to be considered valid

**Severity:** Medium
**Code references:**

- `QueueFactory::_deployQueue`
- `QueueFactory::createQueue`

#### Description

`QueueFactory::createQueue()` charges the deployment fee up-front and then calls `_deployQueue()`, which contains multiple "best-effort" `try/catch` blocks that swallow failures and still returns the newly deployed queue address.

This can lead to a deployed queue being treated as "valid" even though critical integration steps failed, contradicting the expectation that "Queue is fully initialized and ready to use". A practical example is `userRegistry.recordContract` will fail if the deployer has already reached the max number of queues, but since it's in a try/catch block it won't revert and it will just be skipped.

#### Impact

Users can pay the deployment fee and receive a queue that is not actually usable, requiring manual remediation and creating a support burden. In the worst case, a user can be stuck with an irreparably misconfigured queue contract despite having successfully "created" it.

#### Recommendation

Do not ignore errors for steps that are required for core functionality:

- follow Atomicity principles (tx either completes successfully or reverts completely)
- Treat critical external setup calls as mandatory and revert on failure (so the deployment fee can be refunded / no queue is created).

**Notes** fixed in `623e845`

### M-12 - Owner oracle change can OOG as queues grow; no backup path

**Severity:** Medium
**Code references:** `QueueInfo::setGlobalPriceOracle`

#### Description

`QueueInfo::setGlobalPriceOracle()` attempts to "auto-authorize" all existing queues in the new oracle by iterating `queues[]` and calling `setAuthorizedCaller()` per queue:

```
function setGlobalPriceOracle(address _oracle)
    external
    payable
    onlyOwner
    adminAlert
    dependenciesAreSet
{
    // ...
    if (queues.length > 0) {
@>      for (uint i = 0; i < queues.length; i++) {
            try IPriceOracle(_oracle).setAuthorizedCaller(queues[i], true)
             ↪  {
                // Success
            } catch {
                // Skip failed authorizations
            }
        }
    }
    // ...
}
```

As the protocol grows, `queues.length` increases and the single-transaction loop can exceed the block gas limit and the entire call can become permanently unexecutable once the queue list is large enough.

#### Impact

The owner can become unable to upgrade the global oracle (including in incident response scenarios such as oracle bugs, key compromise, or required migrations), leaving the protocol stuck with an outdated or unsafe oracle configuration.

#### Recommendation

Introduce a scalable upgrade path:

- Split the operation into two steps: set the new oracle address (no loop), then authorize queues in batches via a paginated function (e.g., `authorizeQueuesInOracle(uint256 start, uint256 end)`).
- Alternatively, implement a batch authorization method in `PriceOracle` and call it in chunks, or allow queues to self-authorize in the new oracle on first use.

**Notes** fixed in `882322d`

### M-13 - `require(...)` can brick queue if trader loses everything (0 returned value case)

**Severity:** Medium
**Code references:** `AthQueueLib::calculateFinalAmount01`

#### Description

`AthQueueLib::calculateFinalAmount01()` computes the investor's final redeemable amount from `cycleHistory` checkpoints, but enforces that both checkpoints are non-zero:

```
function calculateFinalAmount01(
    IAthQueue.CycleData[] storage cycleHistory,
    uint256 costBasis,
    uint64 entryCycleId
) external view returns (uint256) {
    // ...
    uint256 entryCheckpoint = /* ... */;
    uint256 currentCheckpoint = cycleHistory[cycleHistory.length -
     ↪   1].compoundedRewardPerUnit;

    require(entryCheckpoint > 0, "Entry checkpoint is zero");
@>  require(currentCheckpoint > 0, "Current checkpoint is zero");

    return (costBasis * currentCheckpoint) / entryCheckpoint;
}
```

In a total-loss scenario (or any scenario where `compoundedRewardPerUnit` becomes `0`), the "natural" final amount is `0`, but this `require` forces a revert instead of returning `0`. This makes it impossible to settle/claim in a 0-outcome cycle, and it is particularly relevant if the protocol intends to support "total loss" outcomes (see M-06).

#### Impact

If a 0-checkpoint cycle is ever possible, investor settlement/claims can revert permanently, effectively bricking user exits and/or any lifecycle transitions that rely on final-amount calculations.

#### Recommendation

Handle a 0 checkpoint as a valid "total loss" outcome:

- Replace the `require(currentCheckpoint > 0)` with a branch returning `0` when `currentCheckpoint == 0`.
- Ensure the rest of the queue lifecycle is compatible with a 0-outcome cycle (including checkpointing and any validations in `AthQueue::concludeTradingContract`).

**Notes** fixed in `0632ba3`

### M-14 - Potential OOG revert in looped operation

**Severity:** Medium
**Code references:**

- `QueueInfo::executeTokenChange`
- `QueueInfo::_removeAllowedToken`

#### Description

`QueueInfo::_removeAllowedToken()` removes whitelisted tokens by scanning the full `allowedTokensArray` for each token to remove:

```
function _removeAllowedToken(address[] memory tokens_) internal {
    for (uint256 i = 0; i < tokens_.length; i++) {
        address token = tokens_[i];
        // ...
        if (allowedTokens[token]) {
            allowedTokens[token] = false;

@>          for (uint256 j = 0; j < allowedTokensArray.length; j++) {
                if (allowedTokensArray[j] == token) {
                    allowedTokensArray[j] =
↪   allowedTokensArray[allowedTokensArray.length - 1];
                    allowedTokensArray.pop();
                    break;
                }
            }
        }
    }
}
```

This creates an O(`tokens_.length * allowedTokensArray.length`) loop. As the whitelist grows, token removals can become increasingly expensive and may eventually exceed the block gas limit, making removals impossible.

Since token removal is only reachable through `executeTokenChange(...)` (which calls `_removeAllowedToken(...)`) and there is no alternate admin removal path, an OOG here can make removals impossible in practice.

#### Impact

The owner/admin can be prevented from removing tokens from the global whitelist once it becomes large enough, weakening operational control and incident response (e.g., inability to remove a token in response to an exploit, deprecation, or other emergency).

**Recommendation**

Make removals O(1) per token:

- Maintain an index mapping (e.g., `allowedTokenIndexPlus1[token]`) so you can swap-and-pop without scanning the full array.
- Alternatively, use a standard enumerable set pattern/library that supports constant-time removals.

**Notes** fixed in `9f05ad5`

### L-01 - Hardcoded multisig address with no upgrade path

**Severity:** Low
**Code references:**

- `AthLevel::MULTISIG`
- `AthLevel::emergencyWithdraw`

#### Description

`AthLevel` hardcodes a privileged `MULTISIG` address as a `constant` and uses it for emergency access control:

```
contract AthLevel is Ownable, ReentrancyGuard {
    // ...
@>  address public constant MULTISIG =
 ↪  0x94D68a1b475E7667Bb61F4c739D44F29519dc742;
    // ...

    function emergencyWithdraw(address token, uint256 amount) external {
        require(msg.sender == MULTISIG, "only multisig");
        IERC20(token).safeTransfer(errorWallet, amount);
    }
}
```

If the multisig needs to rotate (key compromise, signers updated, chain migration, operational change), there is no upgrade path inside `AthLevel` to update the privileged address.

#### Impact

This reduces operational resilience: emergency functions can become permanently inaccessible if the hardcoded multisig becomes unusable.

#### Recommendation

Make the multisig address configurable (e.g., set in the constructor and stored in state), and gate updates behind a timelock (and/or a 2-step multisig migration flow).

**Notes** fixed in `93cec8a`

### L-02 - Init can set non-monotone level thresholds, later bricking the setter

**Severity:** Low
**Code references:**

- `AthLevel::constructor`
- `AthLevel::setLevelThreshold`

#### Description

During deployment, level thresholds are loaded without enforcing a strictly increasing progression:

```
constructor(
    IERC20 lpToken_,
    ILpRewards reward_,
    uint256 maxLevel_,
    uint256 unlockDuration_,
    uint256[] memory levelThresholds_,
    address treasuryWallet_,
    address errorWallet_,
    address owner_
) {
    // ...
    for (uint256 i = 0; i < maxLevel_; i++) {
@>      levelThresholdUSD[i + 1] = levelThresholds_[i];
    }
    // ...
}

function setLevelThreshold(uint256 level_, uint256 usdAmount)
    external payable onlyOwner adminAlert
{
    // ...
    if (level_ > 1) {
        require(usdAmount > levelThresholdUSD[level_ - 1], "Threshold must
         ↪  be > previous level");
    }
    if (level_ < maxLevel) {
@>      require(usdAmount < levelThresholdUSD[level_ + 1], "Threshold must
 ↪  be < next level");
    }
    // ...
}
```

If `levelThresholds_` is set to a non-monotone sequence (e.g., `L3 < L2`), the per-level setter can become impossible to use because each change must satisfy both the “previous” and “next” constraints simultaneously. In such a state, an attempted remediation may be blocked by the contract’s own monotonicity checks.

**Impact**

A misconfigured deployment can leave the protocol unable to adjust level thresholds via `setLevelThreshold`, requiring redeployment or more invasive governance actions.

**Recommendation**

Enforce monotonicity in the constructor (strictly increasing thresholds)

**Notes** fixed in `b90487f`

### L-03 - Missing `maxLevelReached` check for active unlock requests after threshold changes

**Severity:** Low
**Code references:**

- `AthLevel::cancelUnlock`
- `AthLevel::_afterLPReceived`

#### Description

`AthLevel` tracks a "highest ever reached" level (`maxLevelReached`), and updates it in the normal deposit flow:

```
function _afterLPReceived(address user_, uint256 lpAmount, uint256
 ↪   usdCredited) private {
    // ...
    uint256 newLevel = _calcLevelFromUSD(u.activeUSD);
    // ...
@>  if (newLevel > u.maxLevelReached) {
        u.maxLevelReached = uint8(newLevel);
    }
    // ...
}
```

However, `cancelUnlock()` recalculates `newLevel` (based on `u.activeUSD`) and updates `u.level`, but does not update `u.maxLevelReached`:

```
function cancelUnlock() external nonReentrant {
    // ...
    uint256 newLevel = _calcLevelFromUSD(u.activeUSD);
    u.level = uint8(newLevel);
    // ...
}
```

If `levelThresholdUSD` values are changed while a user is in the unlock flow, the same `activeUSD` can map to a different `newLevel` at cancel-time. This can lead to an inconsistent state where `u.level > u.maxLevelReached`, until a later operation updates `maxLevelReached` again.

#### Impact

Downstream integrations (or off-chain logic) that rely on `maxLevelReached` can behave incorrectly (e.g., feature gating based on "ever reached" level), and the stored user state can violate the intended invariant `maxLevelReached >= level`.

**Recommendation**

Update `cancelUnlock()` to keep `maxLevelReached` consistent (e.g., `u.maxLevelReached = max(u.maxLevelReached, uint8(newLevel))`).

**Notes** fixed in `c57aa9b`

### L-04 - Deadline passed as constant is ineffective; should be a function parameter

**Severity:** Low
**Code references:** `Zap::zapAndDeposit`

#### Description

`Zap::zapAndDeposit()` hardcodes the router deadlines for both the swap and the add-liquidity steps:

```
function zapAndDeposit(uint256 usdtAmount, address referrer) external
↪   nonReentrant {
    // ...
    uint[] memory amounts = router.swapExactTokensForTokens(
        half,
        minAthxOut,
        path,
        address(this),
@>      block.timestamp + 1200
    );
    // ...
    (,, uint256 liquidity) = router.addLiquidity(
        address(USDT),
        address(ATHX),
        usdtBal,
        athxBal,
        minAmountA,
        minAmountB,
        address(this),
@>      block.timestamp + 1200
    );
    // ...
}
```

since `block.timestamp` is derived at execution time, the deadline check configured in this way is effectively useless as it always passes

#### Impact

weaker user control over time-based MEV exposure.

#### Recommendation

Accept a `deadline` parameter, validate it (e.g., `deadline >= block.timestamp`), and pass it through to router calls.

**Notes** fixed in `4e983ea`

### L-05 - Missing "2nd level" check in referrer setting flows

**Severity:** Low

**Code references:**

- `AthReferral::_addUserReferrer`
- `AthReferral::setReferrerByOwner`
- `AthReferral::_setReferrerInternal`

#### Description

The standard referrer add flow includes a 2-level circular referral check:

```
function _addUserReferrer(address _referrer, address _user) private {
    // ...
    require(referrerOf[_referrer] != _user, "59");

    // Check 2-level circular reference
    address level2 = referrerOf[_referrer];
    if (level2 != address(0)) {
@>      require(referrerOf[level2] != _user, "Circular referral detected");
    }
    // ...
}
```

However, the owner/bulk setting flows do not perform the 2-level check (they only prevent a direct 1-hop cycle):

```
function setReferrerByOwner(address _user, address _referrer, bool _force)
    external payable onlyOwner adminAlert dependenciesAreSet
{
    // ...
@>  require(referrerOf[_referrer] != _user, "Circular referral not
 ↪  allowed");
    // ...
}
```

This means a 2-hop cycle (e.g., `A -> B -> C -> A`) can be introduced via `setReferrerByOwner` / `bulkSetReferrersByOwner`, even if the "normal" referral paths would prevent it.

#### Impact

The referral graph can become cyclic, which can break assumptions in referral accounting, reward distribution, and/or off-chain indexing.

**Recommendation**

Reuse the same validation logic across all referrer-setting entrypoints (e.g., route owner/bulk flows through `_addUserReferrer`).

**Notes** fixed in `b9761bf`

### L-06 - `adminAlert` can be bypassed by calling inherited `transferOwnership` directly

**Severity:** Low
**Code references:** `LpRewards::transferOwnershipWithAlert`

#### Description

`LpRewards` introduces `transferOwnershipWithAlert(...)` as a wrapper that charges `adminAlert`:

```
function transferOwnershipWithAlert(address newOwner) external payable
 ↪  onlyOwner adminAlert {
    require(newOwner != address(0), "New owner is zero address");
@>  transferOwnership(newOwner);
}
```

However, the contract still exposes `Ownable.transferOwnership(...)` directly. As a result, the owner can call `transferOwnership(...)` without paying the alert fee, bypassing the intended `adminAlert` mechanism.

#### Impact

The fee/notification mechanism can be bypassed, weakening the consistency and reliability of "admin alert" expectations.

#### Recommendation

Override `transferOwnership(...)` itself to include `adminAlert` (and optionally remove the wrapper), so there is a single ownership transfer entrypoint with consistent behavior.

**Notes** fixed in `f87ec97`

### L-07 - Missing `errorWallet` existence check

**Severity:** Low
**Code references:** `LpRewards::forceClaimToErrorWallet`

#### Description

`forceClaimToErrorWallet(...)` forwards a user's pending rewards to the `errorWallet` returned by `AthLevel`, but does not validate it is set:

```
function forceClaimToErrorWallet(address user) external onlyLevel
 ↪  nonReentrant {
    _update(user);
    uint256 reward = rewards[user];
    if (reward > 0) {
        rewards[user] = 0;
        address errorWallet = IAthLevel(levelContract).getErrorWallet();
@>      rewardToken.safeTransfer(errorWallet, reward);
        emit FundsHeldForRecovery(user, reward, "Admin force claim");
    }
}
```

This contrasts with `getReward()`, which explicitly checks `errorWallet != address(0)` before using it.

#### Impact

In case of misconfiguration (or unexpected `errorWallet == address(0)`), the recovery flow can send rewards to an inaccessible address, effectively burning the tokens.

#### Recommendation

Add `require(errorWallet != address(0), "errorWallet not set")` before transferring (matching the validation already performed in `getReward()`).

**Notes** fixed in `f677e8f`

### L-08 - Missing `_disableInitializers()` on implementation

**Severity:** Low
**Code references:**

- `AthQueue::constructor`
- `AthQueue::initialize`

#### Description

`AthQueue` is deployed as an implementation contract and then cloned (EIP-1167). The implementation has an empty constructor:

```
// --- Constructor (empty for clone pattern) ---
@>  constructor() {}

function initialize(
    address athLevel_,
    address referralAddress_,
    address userRegistry_,
    address owner_,
    address[] memory initialTraders_,
    address deployer_,
    address queueInfo_,
    address participationToken_,
    address extensionContract_,
    address platformRequirements_
) external {
    require(!_initialized, "Already initialized");
    // ...
}
```

Because `_initialized` is a storage variable, the implementation contract itself starts uninitialized and can be initialized by any caller. While this does not directly affect already-deployed clones, it is a known footgun (implementation can appear as a valid queue, accept funds, and be "owned"/configured by an attacker).

#### Impact

Defense-in-depth issue: increases the chance of deployment/operational mistakes and can be abused to confuse monitoring or user interactions if the implementation address is mistakenly treated as a usable queue.

**Recommendation**

In the implementation constructor, disable initialization (either by setting `_initialized = true` for this custom pattern, or by inheriting OpenZeppelin `Initializable` and calling `_disableInitializers()`).

**Notes** fixed in `8adbdd7`

### L-09 - Inconsistent queue max behavior (QueueInfo caps at 100; here uncapped)

**Severity:** Low
**Code references:**

- `AthQueue::setMaxInvestors`
- `QueueInfo::setMaxInvestorsPerQueue`

#### Description

`QueueInfo` caps the global "max investors per queue" setting to the range `1..100`:

```
function setMaxInvestorsPerQueue(uint8 max_)
    external payable onlyOwner adminAlert dependenciesAreSet
{
@>  require(max_ > 0 && max_ <= 100, "Invalid max");
    maxInvestorsPerQueue = max_;
}
```

However, `AthQueue::setMaxInvestors(uint8 max_)` allows any `uint8` value (including values above `100`) when setting a per-queue override:

```
function setMaxInvestors(uint8 max_) external {
    _onlyOwnerOrDeployer();
@>  maxInvestors = max_;
    emit MaxInvestorsSet(old, max_);
}
```

This creates inconsistent configuration constraints across "global default" vs "per-queue override".

#### Impact

Operational inconsistency and potential downstream assumptions breaking (e.g., UI/monitoring or any logic assuming a maximum of 100).

#### Recommendation

Enforce the same bound in `AthQueue::setMaxInvestors` (e.g., allow `0` for "use global" and otherwise require `max_ <= 100`), unless this is an intended local override.

**Notes** fixed in `6dc1c3b`

### L-10 - Inconsistent behavior vs `invest()`

**Severity:** Low
**Code references:**

- `AthQueueExtension::invest`
- `AthQueueExtension::investFor`

#### Description

`invest()` and `investFor()` are intended to be largely equivalent "entrypoints" into the same investor state machine, but they handle `UserRegistry.recordContract(...)` differently.

In `invest()`, a "new investor" registration attempt is done without any error handling. If `recordContract(...)` reverts (e.g., due to a duplicate record, pause, dependency, or authorization issue inside `UserRegistry`), the entire `invest()` call reverts:

```
function invest(uint256 amount_, address _referrer) external nonReentrant {
    // ...
    if (!isInvestor[investorAddr]) {
        // ...
@>      IUserRegistry(userRegistry).recordContract(
            investorAddr, TYPE_QUEUE, uint64(uint160(address(this))),
 ↪  ROLE_INVESTOR
        );
    }
    // ...
}
```

In `investFor()`, the same registration is wrapped in a `try/catch` and the function continues even if `recordContract(...)` fails:

```
function investFor(uint256 amount_, address beneficiary) external
 ↪  nonReentrant {
    // ...
    if (isNewInvestor) {
        // ...
        try IUserRegistry(userRegistry).recordContract(
            beneficiary, TYPE_QUEUE, uint64(uint160(address(this))),
 ↪  ROLE_INVESTOR
        ) {
            // Success
        } catch {
@>          // Continue even if UserRegistry reverts
```

```
        }
    }
    // ...
}
```

This mismatch makes the "same" user lifecycle behave differently depending on whether funds are deposited via `invest()` or via `investFor()`.

**Impact**

Inconsistent behavior and assumptions:

- `invest()` can revert in situations where `investFor()` succeeds, creating surprising UX and integration behavior.
- If `UserRegistry` records are relied upon for off-chain accounting, compliance, or monitoring, `investFor()` can lead to an investor being funded/active without a corresponding registry record.

**Recommendation**

Unify behavior across both entrypoints, better if both will have no try/catch to ensure atomicity.

**Notes** fixed in `88bdb53`

### L-11 - Router path validation should match both `tokenIn` and `tokenOut`

**Severity:** Low

**Code references:**

- `AthTrader::swap`
- `QueueInfo::tokenRouters`

#### Description

`AthTrader::swap()` selects/validates the router using `tokenRouters(tokenIn)` only:

```
function swap(
    address router_,
    address[] memory route_,
    uint256 amountIn_,
    uint256 amountOutMin_,
    uint256 deadline_,
    bytes calldata priceUpdateData
) external payable onlyTrader nonReentrant {
    // ...
    address tokenIn = route_[0];
    address tokenOut = route_[route_.length - 1];

    // ...

    // Validate provided router matches configured router or use configured
     ↪  one
@>  address configuredRouter = IQueueInfo(queueInfo).tokenRouters(tokenIn);
    // ...
}
```

validating only against `tokenIn` is not enough as the router must support both `tokenIn` and `tokenOut`.

#### Impact

Per-token router controls are only partially enforced, which can make router management less effective and create possible non-valid paths pass the whitelist check.

#### Recommendation

validate both `tokenIn` and `tokenOut` against the given `router`. Also, the passed `router` parameter is actually useless as the only acceptable router is the `configuredRouter`

**Notes** fixed in `2029287`

### L-12 - Unused validation path / whitelist remnants (`validateSwap`)

**Severity:** Low
**Code references:**

- `PriceOracle::onlyAuthorized`
- `PriceOracle::validateSwap`

#### Description

`PriceOracle` defines an access-control mechanism (`authorizedCallers` + `onlyAuthorized`), but the main entrypoint `validateSwap(...)` does not use it:

```
modifier onlyAuthorized() {
@>   require(authorizedCallers[msg.sender] || owner() == msg.sender,
 ↪   "Unauthorized");
     _;
}

function validateSwap(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 expectedOut,
    bytes calldata priceUpdateData
) external payable returns (bool valid, string memory reason) {
    // ...
}
```

This makes `authorizedCallers` effectively unused and leaves `validateSwap` callable by any address. If the intent is for `AthTrader` (or other protocol contracts) to be the only callers, the current implementation does not enforce that intent; if the intent is to allow public calls, the authorization machinery is unnecessary complexity.

#### Impact

Since `validateSwap` is actually fine to be left as public, there is no real immediate impact. However this is clearly a diverge fro the developer's intention and as such it must be considered as a code mistake.

#### Recommendation

Either apply `onlyAuthorized` to `validateSwap` (and any other relevant functions), or remove the unused authorization.

**Notes** fixed in `088b554`

### L-13 - Oracle price confidence is ignored

**Severity:** Low
**Code references:** `PriceOracle::_validateWithPyth`

#### Description

When validating swaps using Pyth, the contract fetches `PythStructs.Price` for each token and converts the returned `price` value into USD terms:

```
function _validateWithPyth(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 expectedOut
) internal returns (bool valid, string memory reason) {
    // ...
    try pythOracle.getPriceNoOlderThan(priceIdIn, maxPriceAge) returns
     ↪  (PythStructs.Price memory priceIn) {
        try pythOracle.getPriceNoOlderThan(priceIdOut, maxPriceAge) returns
         ↪  (PythStructs.Price memory priceOut) {
            uint256 priceInUSD = _convertPythPrice(priceIn);
            uint256 priceOutUSD = _convertPythPrice(priceOut);
            // ...
        }
    }
    // ...
}
```

However, Pyth prices include a confidence interval (`conf`) representing uncertainty in the quoted price. The current validation ignores confidence entirely and treats all prices as equally reliable.

#### Impact

Reduced oracle robustness: low-confidence prices can be used for validation, which weakens the "oracle-based slippage bound" guarantees (especially during volatile markets or oracle degradation).

#### Recommendation

Incorporate confidence into validation (e.g., reject prices where `conf / price` exceeds a configured threshold).

**Notes** fixed in `82a6c09`

### L-14 - “Must not be contract” check can be bypassed via CREATE2 address

**Severity:** Low
**Code references:** `QueueFactory::createQueue`

#### Description

`QueueFactory::createQueue(...)` attempts to enforce that all `authorizedTraders_` are EOAs by checking `extcodesize(trader) == 0`:

```
function createQueue(
    address[] memory authorizedTraders_,
    address participationToken_,
    uint minAmount_,
    uint16 expectedCycleDurationDays_,
    bool useGlobalTokens_,
    address[] memory tradingTokens_
) external payable dependenciesAreSet returns (address) {
    // ...
    for (uint i = 0; i < authorizedTraders_.length; i++) {
        address trader = authorizedTraders_[i];
        uint256 size;
        assembly {
@>          size := extcodesize(trader)
        }
        require(size == 0, "Traders must be EOA");
    }
    // ...
}
```

This check can be bypassed by passing a CREATE2-derived address (or any address where code is not yet deployed). At the time of `createQueue`, `extcodesize` is `0`, but the address can later become a contract with deployed code, undermining the “EOA-only trader” assumption.

#### Impact

The intended “no contract traders” policy is not reliably enforceable.

#### Recommendation

Do not rely on `extcodesize == 0` for EOA verification. Better To directly allow contracts to be trader.

### L-15 - Queue state where it can only trade `participationToken` should be invalid

**Severity:** Low

**Code references:**

- `QueueFactory::_deployQueue`
- `AthQueue::initialize`

#### Description

During deployment, if `useGlobalTokens_ == false` and `tradingTokens_.length == 0`, `_deployQueue()` does not add any extra allowed tokens:

```
function _deployQueue(
    address[] memory authorizedTraders_,
    address participationToken_,
    uint minAmount_,
    uint16 expectedCycleDurationDays_,
    bool useGlobalTokens_,
    address[] memory tradingTokens_
) internal returns (address) {
    // ...
    if (useGlobalTokens_) {
        IAthQueue(clone).setUseGlobalTokensWhitelist(true);
    } else if (tradingTokens_.length > 0) {
        IAthQueue(clone).addToAllowed(tradingTokens_, true);
    }
    // If useGlobalTokens_ == false AND tradingTokens_.length == 0:
@>  // Queue can only trade participationToken (added automatically in
 ↪  initialize)
    // ...
}
```

At the same time, `AthQueue::initialize()` always adds the `participationToken_` to `allowedTokens`, meaning the resulting queue can end up with an "allowed set" containing only the participation token:

```
function initialize(/* ... */ address participationToken_, /* ... */)
 ↪  external {
    // ...
    participationToken = participationToken_;
@>  allowedTokens[participationToken_] = true;
    // ...
}
```

In practice, a queue that can only trade its own participation token is misconfigured and won't be able to perform any trade.

**Impact**

Misconfiguration risk: queues can be deployed into a "valid but unusable" state.

**Recommendation**

require that at least one additional, distinct trading token is provided.

**Notes** fixed in `3221fd3`

### I-01 - Missing array length checks

**Severity:** Info
**Code references:** `AthLevel::bulkImportUsers`

**Description**

`AthLevel::bulkImportUsers(...)` only checks the length of `_users` against `_activeLP`, but then indexes multiple other arrays (`_pendingLP`, `_activeUSD`, `_pendingUSD`, `_unlockTime`, `_level`, `_maxLevelReached`) using the same `i`:

```
function bulkImportUsers(
    address[] calldata _users,
    uint256[] calldata _activeLP,
    uint256[] calldata _pendingLP,
    uint256[] calldata _activeUSD,
    uint256[] calldata _pendingUSD,
    uint256[] calldata _unlockTime,
    uint256[] calldata _level,
    uint256[] calldata _maxLevelReached
) external payable {
    // ...
@>  require(_users.length == _activeLP.length, "Length mismatch");

    for (uint256 i = 0; i < _users.length; i++) {
        // ...
        u.pendingLP = _pendingLP[i];
        u.activeUSD = _activeUSD[i];
        u.pendingUSD = _pendingUSD[i];
        u.unlockTime = _unlockTime[i];
        u.level = uint8(_level[i]);
        u.maxLevelReached = uint8(_maxLevelReached[i]);
        // ...
    }
}
```

If any of the other arrays has a different length than `_users`, the call will revert due to out-of-bounds reads.

**Impact**

Operational/UX issue during migration: the migrator/owner can accidentally brick a migration batch transaction due to input shape mistakes.

**Recommendation**

Add explicit length checks for all arrays (or accept a single array of structs to avoid parallel arrays entirely).

**Notes** fixed in `10078f0`

### I-02 - Zap 10% cap may be too low for low-cap tokens

**Severity:** Info
**Code references:** `Zap::setSlippageBps`

#### Description

`Zap::setSlippageBps(...)` hard-caps slippage to `<= 1000` bps (10%):

```
function setSlippageBps(uint256 bps) external payable onlyOwner adminAlert
 ↪  {
@>  require(bps <= 1000, "slippage too high");
    slippageBps = bps;
    emit SlippageSet(bps);
}
```

Depending on the token pair liquidity and volatility, 10% may be too restrictive and could cause zaps to revert even when users are willing to accept higher slippage.

#### Impact

Operational limitation: some deposits via zap may become unusable for low-liquidity / high-volatility pairs.

#### Recommendation

Re-evaluate the 10% ceiling and/or make the cap configurable (with clear safeguards, since higher slippage increases MEV/execution risk).

**Notes** fixed in `c9d0d76`

### I-03 - Docs: child auth can authorize arbitrary contracts (no direct impact)

**Severity:** Info
**Code references:**

- `AthReferral` docs/comment‘
- `AthReferral::authorizeContract`

#### Description

`AthReferral::authorizeContract(...)` is documented as allowing authorized factories to authorize their “children”, but there is no enforcement that `_contract` is a child deployed by `msg.sender` (or even that it is a contract):

```
function authorizeContract(address _contract) external {
    require(
        authorizedContracts[msg.sender],
        "Not authorized factory"
    );
    require(_contract != address(0), "Invalid address");
@>  authorizedContracts[_contract] = true;
}
```

In practice, any already-authorized contract can authorize any address as `authorizedContracts[_contract] = true`.

#### Impact

Documentation/expectation mismatch: integrators and reviewers may assume a stronger “factory -> child only” authorization invariant than what is implemented.

#### Recommendation

Either update the documentation to match the actual semantics (trusted authorized contracts can authorize additional contracts), or enforce a verifiable factory-child relationship (e.g., require authorization to be performed by a single admin role, or add a factory registry + on-chain provenance checks).

**Notes** fixed in `8a5ba4c`

### I-04 - Incorrect comment: owner is QueueFactory admin

**Severity:** Info

**Code references:**

- `AthQueue::trader`
- `QueueFactory::_deployQueue`

#### Description

`AthQueue::trader()` returns `owner()`:

```
function trader() external view override returns (address) {
@>   return owner();
}
```

However, queues are initialized by `QueueFactory` with `owner_ = owner()` (the `QueueFactory` owner/admin), not the queue deployer:

```
try IAthQueue(clone).initialize(
    address(athLevel),
    referralAddress,
    userRegistry,
@>  owner(),               // owner (admin)
    authorizedTraders_,
    msg.sender,            // deployer
    // ...
) {
    // ...
}
```

As a result, `AthQueue::trader()` effectively returns the global QueueFactory admin, which can be confusing given the “trader” naming and associated comments.

#### Impact

Readability/operational risk: tooling, integrators, or reviewers may misinterpret the meaning of `trader()` (e.g., expecting the deployer/manager EOA rather than the global admin).

#### Recommendation

Clarify the role in code/comments (e.g., rename in the interface to `admin()` / `queueOwner()` or explicitly document that `trader()` returns the global QueueFactory admin, while actual trading is executed by `traderContract` and gated by `authorizedTraders`).

**Notes** fixed in `ff0e412`

### I-05 - `uint8` limits investors to 256

**Severity:** Info

**Code references:**

- `AthQueue::maxInvestors`
- `AthQueue::setMaxInvestors`

#### Description

`AthQueue::maxInvestors` is stored as a `uint8`, which caps the per-queue investor limit to `<= 255` (with `0` used as "use global from QueueInfo"):

```
/// @notice Max investors for this queue (0 = use global from QueueInfo)
@>  uint8 public maxInvestors;
```

#### Impact

Scalability/UX limitation: queues cannot support more than 255 investors even if the protocol wants to raise the cap.

#### Recommendation

Use a wider type (`uint16`/`uint32`) for `maxInvestors` (and keep the "0 means use global" sentinel if desired).

**Notes** no fix, sponsor states that system is assumed to work with <= 100 investors so uint8 will be more than enough.

### I-06 - Trader can start trading with `standbyAmount == 0`

**Severity:** Info
**Code references:**

- `AthQueueExtension::startTradingContract`
- `AthQueue::concludeTradingContract`

#### Description

`AthQueueExtension::startTradingContract()` starts a cycle unconditionally and sets `totalTradingAmount = totalStandyAmount` without enforcing that `totalStandyAmount > 0`:

```
function startTradingContract() external {
    // ...
    currentState = IAthQueue.QueueState.trading;
    // ...
@>  totalTradingAmount = totalStandyAmount;
    totalStandyAmount = 0;

    IERC20(participationToken).safeTransfer(traderContract,
↪   totalTradingAmount);
    // ...
}
```

If `totalStandyAmount == 0`, the queue transitions into `trading` with `totalTradingAmount == 0`. This can later make `concludeTradingContract(...)` revert due to `NoFundsInTrading()`:

```
function concludeTradingContract(uint256 returnFromCycle_) external
↪   nonReentrant {
    // ...
@>  if (totalTradingAmount == 0) revert NoFundsInTrading();
    // ...
}
```

#### Impact

Operational footgun: a trading cycle can be started with no funds, potentially causing a stuck/invalid trading state depending on the available recovery paths.

**Recommendation**

Add a guard such as `require(totalStandyAmount > 0, "No funds");` before switching to `trading` (or keep the state in `standby` when the amount is `0`).

**Notes** fixed in `9a93e09`

### I-07 - View price update calls may revert; function mainly for integrators

**Severity:** Info
**Code references:** `PriceOracle::estimateSwapOutput`

#### Description

`PriceOracle::estimateSwapOutput(...)` is a `view` helper that calls into Pyth for price reads and reverts on failure:

```
function estimateSwapOutput(
    address tokenIn,
    address tokenOut,
    uint256 amountIn
) external view returns (uint256 expectedOut, uint256 minWithSlippage) {
    // ...
    try pythOracle.getPriceNoOlderThan(priceIdIn, maxPriceAge) returns
     ↪  (PythStructs.Price memory priceIn) {
        // ...
    } catch {
@>      revert("Cannot fetch input token price");
    }
}
```

This behavior is reasonable, but integrators may mistakenly assume this helper always succeeds (e.g., even if the feed is stale).

#### Impact

Integration UX issue: off-chain callers and frontends may see unexpected reverts when prices are stale/unavailable.

#### Recommendation

Consider providing a non-reverting variant (e.g., `tryEstimateSwapOutput(...) returns (bool ok, uint256 expectedOut, uint256 minWithSlippage)`) for integrators, or document explicitly that this function may revert when oracle reads fail/stale.

**Notes** fixed in `e95c783`

### I-08 - Missing essential deps in config/event (`impl` and `Ext`)

**Severity:** Info
**Code references:**

- `QueueFactory` storage‘
- `QueueFactory::setImplementation` / `QueueFactory::setExtension`
- `QueueFactory::setDependencies`

#### Description

`QueueFactory::setDependencies(...)` marks `dependenciesSet = true`, but does not set (nor emit) the implementation and extension addresses used for queue deployment (those are configured via separate calls):

```
function setDependencies(
    IAthLevel _athLevel,
    address _referralAddress,
    address _userRegistry,
    address _queueInfo,
    address _platformRequirements
) external onlyOwner {
    // ...
    dependenciesSet = true;
    emit DependenciesSet(address(_athLevel), _referralAddress,
     ↪  _userRegistry, _queueInfo);
}
```

This makes the configuration surface harder to reason about from events alone, and can lead to partial configuration states (e.g., deps set but `athQueueImplementation` / `athQueueExtension` not yet configured).

#### Impact

Operational/readability issue: deployment and monitoring tooling may miss critical configuration fields if it only listens to `DependenciesSet`, and operators can end up in a “partially configured” state.

#### Recommendation

Consider emitting a single, complete configuration event (including `impl`, `Ext`, and `platformRequirements`) or provide a dedicated “configuration finalized” event that is only emitted once all required addresses (including implementation/extension) are set.

**Notes** fixed in `e63d135`

## I-09 - Consider `nonReentrant` due to external call to sender

**Severity:** Info
**Code references:** `QueueFactory::createQueue`

### Description

`QueueFactory::createQueue(...)` performs external calls (fee transfer and refund) via `.call{value: ...}("")`:

```
function createQueue(/* ... */) external payable dependenciesAreSet returns
↪   (address) {
    // ...
    (bool success, ) = treasury.call{value: traderDeploymentFee}("");
    require(success, "Fee transfer failed");
    // ...
    if (msg.value > traderDeploymentFee) {
@>      (bool success, ) = msg.sender.call{value: msg.value -
↪   traderDeploymentFee}("");
        require(success, "Refund failed");
    }
    // ...
}
```

While the function currently enforces `tx.origin == msg.sender` (making reentrancy via the refund path practically infeasible for contracts), the pattern is still generally safer with a reentrancy guard (especially if this EOA restriction ever changes).

### Impact

Defense-in-depth: reduces risk of future refactors introducing reentrancy hazards around fee/refund logic.

### Recommendation

Consider adding `nonReentrant` (and/or switching to a pull-based refund pattern) to harden the external call flow.

**Notes** fixed in 8b67c95

### I-10 - Dead code: `lifetimeFeePaid` never incremented

**Severity:** Info
**Code references:**

- `IAthQueue::InvestorData::lifetimeFeePaid`
- `AthQueueLib::calculateLifetimeFees`
- `AthQueueExtension::_processClaim`

#### Description

The `InvestorData` struct contains `lifetimeFeePaid`, and `AthQueueLib::calculateLifetimeFees(`
uses it to compute incremental fees:

```
uint256 totalFee = _calculateFee(lifetimeProfit, investorAddr, athLevel,
 ↪   queueInfo);
uint256 newFee = totalFee > investor.lifetimeFeePaid
    ? totalFee - investor.lifetimeFeePaid
    : 0;
```

However, `lifetimeFeePaid` is not incremented anywhere in the codebase; it is only reset to `0` (e.g., after claim):

```
function _processClaim(address investorAddr, bool forceErrorWallet) private
 ↪   {
    // ...
@>  investor.lifetimeFeePaid = 0;
    // ...
}
```

This makes the “incremental fee” logic effectively unused.

#### Impact

Maintainability issue: dead or misleading state increases cognitive load and can lead to incorrect assumptions during future changes.

#### Recommendation

Either remove `lifetimeFeePaid` (and simplify the fee calculation), or correctly maintain it (e.g., update `investor.lifetimeFeePaid` when fees are assessed) if partial/stepwise fee accounting is intended.

**Notes** fixed in `e33b5ae`

### I-11 - Fee level cannot be set to 0 due to `require`

**Severity:** Info
**Code references:** `AthQueueLib::_calculateFee`

#### Description

`AthQueueLib::_calculateFee(...)` disallows a configured `0`% fee by requiring `feePercent > 0`:

```
function _calculateFee(
    uint256 profitAmount,
    address investorAddr,
    IAthLevel athLevel,
    address queueInfo
) internal view returns (uint256) {
    // ...
    uint8 feePercent = IQueueInfo(queueInfo).athLevelFee(level);
@>  require(feePercent > 0, "Investor fee not configured for level");
    return (profitAmount * feePercent) / 100;
}
```

#### Impact

Configuration limitation: the protocol cannot intentionally set a `0`% fee for a level (e.g., a promotional tier or fee exemption) without code changes.

#### Recommendation

Allow `feePercent == 0` as a valid configuration (and use a separate mechanism to differentiate "unset" vs "set to 0" if needed).

**Notes** fixed in `8405114`

### I-12 - Minimum fee effectively 1%, potentially too high

**Severity:** Info
**Code references:** `AthQueueLib::_calculateFee`

#### Description

Fees are expressed as integer percentages (`uint8 feePercent`) and applied with division by `100`:

```
uint8 feePercent = IQueueInfo(queueInfo).athLevelFee(level);
return (profitAmount * feePercent) / 100;
```

This implies fee granularity of `1%` and cannot represent sub-percent fees (e.g., 25 bps / 0.25%). Combined with the `feePercent > 0` requirement, the effective minimum configurable fee becomes `1%`.

#### Impact

Economic/configuration limitation: fees may be higher than desired for some products or market conditions, and cannot be tuned precisely.

#### Recommendation

Consider representing fees in basis points (`uint16`/`uint32`) and dividing by `10_000`, enabling finer-grained configurations while keeping the same overall design.

**Notes** fixed in `79eaca6`

## General recommendations

These are design, best practices, future-proofing advices that are not derived from specific lines of code but rather from the general structure of the current project. They do not pose an immediate risk and as such are not required to be immediately changed, or changed at all; nonetheless, the SR identified them as *objective* improvements that would benefit the protocol if implemented.

- **Remove redundant overflow checks (Solidity `>=0.8.x`):** Most arithmetic operations already revert on overflow/underflow by default. Extra “max safe” pre-checks increase gas and can introduce additional failure modes if the bounds are overly conservative.
- **Revisit the `adminAlert` mechanism:** Requiring ETH to be sent on every admin call (and forcing an immediate transfer to `errorWallet`) adds a brittle dependency where admin operations can be DOSed if the downstream transfer fails (misconfiguration, revert, gas grief, etc.). Prefer emitting alert events and monitoring off-chain via any 3rd party that offer these kind of services.
- **Standardize blacklist behavior:** Blacklist checks are not consistent across the system (some flows hard-revert if `userRegistry` is unset, others effectively treat it as “not blacklisted”). Pick one invariant (e.g., “dependency must be set” or “unset disables checks”) and apply it uniformly across all entry points to avoid surprising behavior and integration mistakes.
- **Improve zap sandwich/MEV protection:** Heuristic “anti-sandwich” approaches tend to be bypassable and can create false positives, especially on low-liquidity pairs. The typical pattern is to have the caller pass `amountOutMin`/`minLiquidity` parameters (computed off-chain using DEX quotes) so the user explicitly controls slippage bounds and the transaction is protected by deterministic minimums.
- **Prefer custom errors over revert strings:** Replacing `require(..., "message")` with custom errors reduces deployment/runtime gas and makes revert handling more structured for integrators.
- **Consider moving referrals off-chain (event-based):** Persisting referral relationships and accounting fully on-chain can become expensive at scale and is not always a strong decentralization requirement. A common alternative is to emit canonical events and maintain referral state off-chain.
- **Strengthen the testing strategy:** The current tests appear closer to scripts than a fast, repeatable unit/integration suite. A practical goal is a deterministic local test suite (e.g., Foundry) that runs on every change and covers core lifecycle, permissioning, and accounting invariants.
- **Avoid using `try/catch` blocks:** Smart contracts are by default Atomic: changes either fully succed or they fully revert. `try/catch` blocks break this pattern and make transactions hard

to reason/validate as it adds multiple possible paths, try avoiding them and prefer explicit optionality.

- **Define roles and trust boundaries more rigidly:** Access control is currently difficult to reason about across modules. Consolidate role definitions and document them clearly (e.g., Owner/Admin/Deployer/AuthorizedTrader/Bot), and enforce them consistently using a standard pattern (explicit modifiers or a unified access-control module).