



X



ALIEN BASE

Security Review Report for Alien Base

April 2026



Table of Contents

1. About Hexens

2. Executive summary

3. Security Review Details

- Security Review Lead
- Scope
- Changelog

4. Severity Structure

- Severity characteristics
- Issue symbolic codes

5. Findings Summary

6. Weaknesses

- Attacker can register order on behalf of any maker and bypass signature validation, enabling theft of approved token
- Keeper Should Be Included in the Order Struct
- Canceled DCA Orders Can Be Reinitialized with Old Signatures
- Completed DCA order can be replayed indefinitely
- The total filled amount of a DCA order could be greater than the order's totalAmountIn
- inverse Trigger Condition in `_isOrderReady` Leads to Premature Execution
- TWAP deviation check compares square root prices instead of prices
- TWAP oracle source can change after route execution
- Oracle slippage check uses gross input instead of fee-adjusted amount
- `FillOrder` Accepts all Order Types
- Lack of slippage checks for makers in `matchOrders` and `batchMatchOrders` functions
- Fallback oracle prices are used without consistent validation
- Missing Order Parameter Validation in `executeDcaOrder` function
- `executeTrusted()` uses stale `_currentKeeperFeeBps` from previous action of Router
- Order Type Should Be Included in Order Struct

- Redstone payload freshness check uses inconsistent timestamp units
- Surplus Taken Before Fees Causes Inverted Incentives
- Incorrect Permit2 interface will causes permit2 to be permanently broken
- Order cancellation does not validate owner on Indexer side
- TWAP price limits the tick price to 128-bit
- Inconsistent zero-slippage handling in ring matcher paths
- Hybrid overflow keeper fees are sent to the matcher contract
- DCA shortCircuit parameter is not propagated by the keeper
- ExecuteTrusted lacks guard against Stop-Loss/DCA fills
- Swap Surplus Logic Is Bypassable and Economically Flawed
- Post-hook receives gross DEX output, not net user amount
- minExecutionValue Does Not scale with decimals
- Matcher fills can desynchronize indexed order status
- Referral Fee Deducted but Not Distributed When referrer Is Zero Address
- Inconsistent Fee Rounding Can Cause False Reverts
- /api/keeper/pending Can be blocked by unsupported or phantom signed rows
- Mismatched setProtocolFee Parameters Between Manager and Router
- Bit-Invalidator cancellations are not reflected in off-chain order status
- Redundant Order Creation in executeTrailingStop()
- Redundant oraclePrice == 0 Check
- Reuse remaining instead of recalculating completion condition
- Stop-Loss orders with maxslippageBps Above 5000 Will Revert with Underflow

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for Alienbase Epsilon protocol.

Our security assessment was a full review of the code, spanning a total of 1.5 weeks.

During our review, we identified two critical and three high severity vulnerabilities that could have resulted in major disruption and potential asset loss.

We also identified several minor severity vulnerabilities.

All reported issues were fixed or acknowledged by the development team and subsequently verified by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

- Review Led by

Kasper Zwijsen, Head of Audits

- Scope

The analyzed resources are located on:

 <https://github.com/alienbase-xyz/Epsilon/tree/d5545576a098dbf7a979e9618eb5278d2b22d896>

The issues described in this report were fixed in the following commit:

 <https://github.com/alienbase-xyz/Epsilon/tree/8b8be152e8791753ba041522f27134e06e6295b3>

- Changelog

	27 April 2026	Audit start
	7 May 2026	Initial report
	19 May 2026	Revision received
	1 June 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

▪ Severity Characteristics

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium

Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.

Low

Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.

Informational

Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

▪ Issue Symbolic Codes

Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

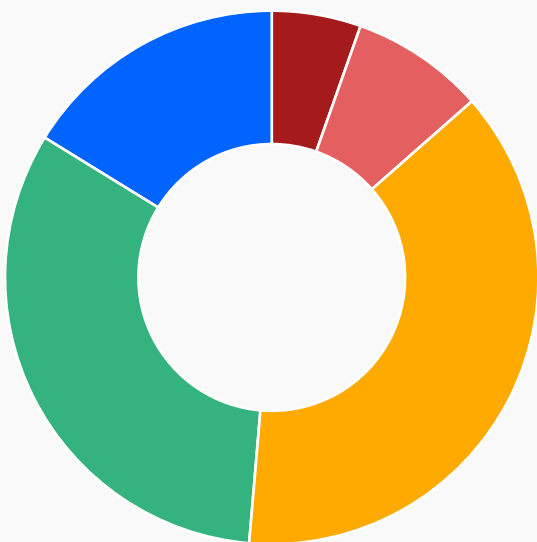
Severity

Number of findings

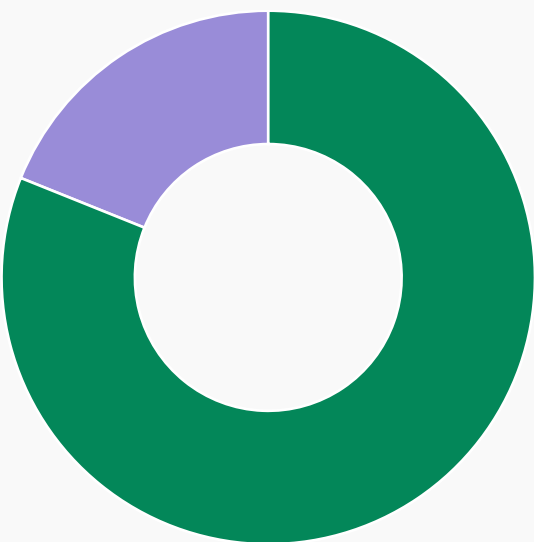
Critical	2
High	3
Medium	14
Low	12
Informational	6

Total:

37



- Critical
- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

6. Weaknesses

This section contains the list of discovered weaknesses.

ALIEN-5 | Attacker can register order on behalf of any maker and bypass signature validation, enabling theft of approved token

Fixed 

Severity:

Critical

Probability:

Very likely

Impact:

Critical

Path:

contracts/EpsilonRouter.sol#L747-L755

Description:

`registerOrder()` allows any address to register an order for any maker, there is no check to validate `msg.sender == order.maker`.

```
function _registerBase(Order calldata order, address _keeper) internal returns (bytes32 orderHash) {
    _validateOrderParams(order);
    orderHash = hashOrder(order);
    if (orderData[orderHash].exists) revert AlreadyRegistered();
    orderData[orderHash].exists = true;
    _orderCreator[orderHash] = order.maker;
    if (_keeper != address(0)) { designatedKeeper[orderHash] = _keeper; emit DesignatedKeeperSet(orderHash, _keeper); }
    emit OrderRegistered(orderHash, order.maker);
}
```

Therefore, an attacker can register and order with a victim's address as maker and call `fillOrder` using a null signature to steal that victim's funds. After that, `fillOrder` function will trigger `_validateMatchSig` function but not verify the signature. Because when order is existed (!`orderData[hash].exists` is true), `_verifySignature` is only triggered when signature length > 0.

```
function _validateMatchSig(address maker, bytes32 hash, bytes calldata sig) private view {
    if (!orderData[hash].exists && sig.length > 0) _verifySignature(maker, hash, sig);
    else if (!orderData[hash].exists) revert BadSignature();
}
```

The vulnerability also exists in the **registerDcaOrder** + **executeDcaOrder** and **registerTrailingStopOrder** + **executeTrailingStop** flows. This is because the **registerDcaOrder** and **registerTrailingStopOrder** functions don't verify that the sender is the maker, as well as **executeDcaOrder** and **executeTrailingStop** do not verify the signature when an order existed.

Attack scenario:

1. Victim has approved the EpsilonRouter for WETH (standard for any protocol user).
2. Attacker deploys a contract that atomically:
 - Constructs an **Order** with **maker = victim**, **receiver = attacker**, **referrer = attacker**, **referralFeeBps = maxReferralFeeBps**, **triggerPrice = 0**, **amountIn = victim's WETH balance**.
 - Calls **registerOrder(order, attackerContract)**, this sets **orderData[hash].exists = true** and **designatedKeeper[hash] = attackerContract**.
 - Calls **fillOrder(order, "", routes, maxKeeperFeeBps)** with routes that swap through an approved DEX.
3. The victim loses all approved WETH. The attacker receives the full swap output minus protocol fee.

Remediation:

Consider to enforce caller identity in **_registerBase()**:

```
function _registerBase(Order calldata order, address _keeper) internal returns (bytes32 orderHash) {
+   if (msg.sender != order.maker) revert Unauthorized();
   _validateOrderParams(order);
   orderHash = hashOrder(order);
   if (orderData[orderHash].exists) revert AlreadyRegistered();
   orderData[orderHash].exists = true;
   _orderCreator[orderHash] = order.maker;
   if (_keeper != address(0)) { designatedKeeper[orderHash] = _keeper; emit DesignatedKeeperSet(orderHash,
   _keeper); }
   emit OrderRegistered(orderHash, order.maker);
}
```

ALIEN-9 | Keeper Should Be Included in the Order Struct

Fixed ✓

Severity:

Critical

Probability:

Likely

Impact:

Critical

Path:

contracts/EpsilonRouter.sol#L427

Description:

Currently, the **Order** struct does not include a field for the designated keeper, and **_verifySignature()** only validates the signature against the order hash without taking the designated keeper into account.

As a result, an attacker can create the order on behalf of the user while assigning a malicious keeper.

Specifically, suppose a user intends to create a trailing stop order with the designated keeper set to **OxKeeper**. An attacker can front-run this by calling **updateTrailingStopHighWaterMark()** with **params.designatedKeeper = OxAttacker**. At line 487, the contract assigns the keeper as follows:

```
designatedKeeper[orderHash] = params.designatedKeeper;
```

Consequently, the attacker becomes the designated keeper and gains control over executing the order instead of the intended keeper.

In the worst case, when the order's **trustKeeper** trait is **True**, the swap validation will be skipped.

```
function _shouldSkipValidation(Order calldata order, bytes32 orderHash) internal view returns (bool) {
    if (!order.makerTraits.trustKeeper()) return false;

    address keeper = designatedKeeper[orderHash];
    // Only skip if trustKeeper is set AND caller is the designated keeper
    return keeper != address(0) && msg.sender == keeper;
}
```

The attacker can utilize this to set the malicious swap data **SwapRoutes.callDatas** (eg: setting the receiver of the swap to attacker's address) to steal all the order's amount.

Remediation:

Consider adding a new field for the designated keeper for the order.

```
struct Order {  
    uint256 salt;  
    address maker;  
    address receiver;  
    address tokenIn;  
    address tokenOut;  
    uint256 amountIn;  
    uint256 triggerPrice;  
    MakerTraits makerTraits;  
    address referrer;  
    uint16 referralFeeBps;  
++    address designatedKeeper;  
}
```

ALIEN-6 | Canceled DCA Orders Can Be Reinitialized with Old Signatures

Fixed 

Severity:

High

Probability:

Likely

Impact:

High

Path:

contracts/EpsilonRouter.sol#L732

Description:

`cancelOrder` marks remaining-based orders as fully filled, then deletes the stored order state and designated keeper when the caller is the order creator.

```
else {
    _remainingInvalidator[msg.sender][orderHash] = RemainingInvalidatorLib.fullyFilled();
    emit OrderCancelled(orderHash);
}
if (_orderCreator[orderHash] == msg.sender) {
    delete orderData[orderHash];
    delete designatedKeeper[orderHash];
    delete orderPostHook[orderHash];
    delete _orderCreator[orderHash];
}
```

However, the lazy initialization path in `executeDcaOrder` does not check `_remainingInvalidator` before accepting a previously valid maker signature and recreating `orderData`.

```
if (data.slot2 == 0 && !data.exists) {
    _verifySignature(maker, orderHash, signature);
    if (!order.makerTraits.allowMultipleFills()) revert NotDcaOrder();
    if (params.intervalSeconds == 0 || params.totalExecutions == 0) revert InvalidParams();
    if (params.flexibilityBps > 2000) revert InvalidParams();

    uint256 baseAmount;
    unchecked {
        baseAmount = params.totalAmountIn / params.totalExecutions;
    }
    if (baseAmount != order.amountIn) revert DcaAmountMismatch();
}
```

```

unchecked {
    data.slot1 = (uint256(params.totalExecutions) << 128) | uint256(params.intervalSeconds);
}
data.exists = true;
_orderCreator[orderHash] = maker;
data.totalAmountIn = params.totalAmountIn;
data.flexibilityBps = params.flexibilityBps;
data.shortCircuit = params.shortCircuit;
}

```

The DCA schedule parameters are also not part of **hashOrder**, so the old signature authorizes only the base order fields, while the DCA execution parameters are supplied by the caller.

```

return _hashTypedData(keccak256(abi.encode(
    ORDER_TYPEHASH,
    order.salt,
    order.maker,
    order.receiver,
    order.tokenIn,
    order.tokenOut,
    order.amountIn,
    order.triggerPrice,
    MakerTraits.unwrap(order.makerTraits),
    order.referrer,
    order.referralFeeBps
)));

```

As a result, a keeper holding an old valid DCA signature can reinitialize and execute a **canceled DCA** order while the order remains unexpired and the maker still has sufficient balance and allowance.

Remediation:

Make **executeDcaOrder** check the existing invalidation state before lazy-initializing or executing a DCA order, and reject orders that were already canceled or fully filled. Also ensure DCA-specific parameters are bound to the maker's signed authorization or to an authenticated registration flow, so old signatures cannot recreate canceled schedules with caller-supplied parameters.

ALIEN-4 | Completed DCA order can be replayed indefinitely

Fixed 

Severity:

High

Probability:

Likely

Impact:

High

Path:

contracts/EpsilonRouter.sol:executeDcaOrder#L401-L405

Description:

When a DCA order completes all executions, **executeDcaOrder()** deletes the order's storage but does not invalidate the maker's signature. Because the registration check only depends on this deleted storage, a keeper can replay the original calldata to restart the order from the beginning, repeatedly draining the maker's tokens beyond the intended **totalAmountIn**.

Specifically, on completion, the function **executeDcaOrder()** clears **orderData[orderHash]** and **_orderCreator[orderHash]**:

```
if (executionsCompleted + 1 >= storedTotalExecutions) {  
    delete orderData[orderHash];  
    delete _orderCreator[orderHash];  
}
```

This deletion resets **orderData[orderHash].exists** to **false** and **slot2** to 0. The registration guard at line 337 will then accept any order that satisfies these two conditions:

```
if (data.slot2 == 0 && !data.exists) {  
    _verifySignature(maker, orderHash, signature);  
    ...  
    data.exists = true;  
}
```

Since the maker's signature is tied to the immutable **orderHash** (derived from the order parameters), and no nonce or invalidation mechanism is consumed for DCA orders — unlike other order types which call **_invalidateOrder** before completion — the original signature remains valid indefinitely.

All other execution paths (regular, trailing stop, batch orders) call **_invalidateOrder**, which marks the nonce as used via **_bitInvalidator** or decrements **_remainingInvalidator**. The DCA flow does neither, leaving it replayable even after completion.

```
unchecked {  
    uint256 remaining = storedTotalExecutions - (executionsCompleted + 1);  
    emit OrderFilled(orderHash, maker, msg.sender, makingAmount, amountOut, remaining);  
    if (executionsCompleted + 1 >= storedTotalExecutions) { delete orderData[orderHash]; delete  
_orderCreator[orderHash]; }  
}
```

Remediation:

Consider invalidating the maker's signature upon DCA completion by calling `_invalidateOrder` before deleting storage.

ALIEN-3 | The total filled amount of a DCA order could be greater than the order's totalAmountIn

Fixed 

Severity:

High

Probability:

Likely

Impact:

High

Path:

contracts/EpsilonRouter.sol:_calculateDcaAmount#L798-L823

Description:

The function `_calculateDcaAmount` is used to calculate the filled amount given the total executions `totalExecutions` and the number of successful executions `executionsCompleted`.

```
function _calculateDcaAmount(
    OrderData storage data,
    uint128 executionsCompleted,
    uint128 totalExecutions
) internal view returns (uint256) {
    uint256 totalAmountIn = data.totalAmountIn;
    uint256 baseAmount;

    unchecked {
        baseAmount = totalAmountIn / totalExecutions;
    }
    unchecked {
        if (executionsCompleted + 1 == totalExecutions) {
            return totalAmountIn - (baseAmount * executionsCompleted);
        }
    }

    uint16 flexibilityBps = data.flexibilityBps;
    if (flexibilityBps > 0) {
        uint256 variation;

        unchecked {
            variation = (baseAmount * flexibilityBps) / BPS_DENOMINATOR;
            return executionsCompleted % 2 == 0
                ? baseAmount + variation
                : (baseAmount > variation ? baseAmount - variation : baseAmount);
        }
    }
}
```

```

    }
    return baseAmount;
}

```

Lines 806–808 show that the filled amount of the last execution is the remaining input amount of the order, which is **totalAmountIn - (baseAmount * executionsCompleted)**. This return value assumes that after **executionsCompleted** executions, the already filled amount is **baseAmount * executionsCompleted**.

However, this assumption is incorrect due to the introduction of **flexibilityBps**. The **flexibilityBps** is applied to the **baseAmount** such that in even executions, the filled amount is **baseAmount + variation**, while in odd executions, the filled amount is **baseAmount - variation**.

As a consequence, by the last execution, the already filled amount could be **baseAmount * executionsCompleted + variation** if **totalExecutions** is an even number.

Example:

Consider the following example where **totalExecutions = 4** and **flexibilityBps > 0**:

1. First fill: **baseAmount + variation**
2. Second fill: **baseAmount - variation**
3. Third fill: **baseAmount + variation**
4. Fourth fill (last execution): the condition at lines 806–808 is triggered, returning **totalAmountIn - (baseAmount * 3)**

--> The total filled amount will be:

```

(baseAmount + variation)
+ (baseAmount - variation)
+ (baseAmount + variation)
+ (totalAmountIn - (baseAmount * 3))
= totalAmountIn + variation

```

This is greater than the order's input amount **totalAmountIn**.

As a consequence, the maker of the order will be filled more than expected.

Remediation:

Consider taking into account whether **totalExecutions** is even or odd when calculating the final execution amount.

ALIEN-34 | inverse Trigger Condition in `_isOrderReady` Leads to Premature Execution

Acknowledged

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

`./contracts/epsilonRouter.sol`

Description:

The `_isOrderReady` function applies the tolerance (**band**) in the wrong direction for both stop-loss and take-profit orders. As a result, orders can be executed before the intended trigger condition is actually met.

```
function _isOrderReady(Order calldata order, uint256 currentPrice) internal view returns (bool) {
    if (order.triggerPrice == 0) return true;
    uint256 band = (order.triggerPrice * triggerToleranceBps) / BPS_DENOMINATOR;

    if (order.makerTraits.isStopLoss()) {
        return currentPrice <= order.triggerPrice + band;
    } else {
        return currentPrice + band >= order.triggerPrice;
    }
}
```

The tolerance is meant to create a buffer zone around the trigger price, ensuring execution only happens after a meaningful move reducing noise.

However, the current logic instead expands the trigger condition inward, allowing execution within the tolerance band, not beyond it.

Example

- `triggerPrice = 100`
- `triggerToleranceBps = 300 (3%)`
- `band = 3`

Stop-Loss Case

Current logic:


```
currentPrice <= triggerPrice + band  
currentPrice <= 100 + 3  
currentPrice <= 103
```

- `currentPrice = 102`

```
102 <= 103 → true
```

The order executes, even though price is still above the trigger price (100).

Remediation:

Apply the tolerance away from the trigger price, not toward it:

```
function _isOrderReady(Order calldata order, uint256 currentPrice) internal view returns (bool) {  
    if (order.triggerPrice == 0) return true;  
  
    uint256 band = (order.triggerPrice * triggerToleranceBps) / BPS_DENOMINATOR;  
  
    if (order.makerTraits.isStopLoss()) {  
        return currentPrice <= order.triggerPrice - band;  
    } else {  
        return currentPrice >= order.triggerPrice + band;  
    }  
}
```

ALIEN-38 | TWAP deviation check compares square root prices instead of prices

Fixed 

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

contracts/EpsilonTWAPOracle.sol#L247

Description:

`EpsilonTWAPOracle` is intended to limit the deviation between the current spot price and the TWAP price using `maxDeviationBps`. However, the deviation check compares Uniswap V3 `sqrtPriceX96` values directly instead of comparing the underlying prices.

```
uint160 sqrtPriceX96TWAP = _getSqrtPriceX96TWAP(poolContract);
uint256 deviation = _calculateDeviationBps(sqrtPriceX96Spot, sqrtPriceX96TWAP);
if (deviation > maxDeviationBps) revert TWAPDeviationTooHigh();
```

The helper calculates the percentage difference between the two square root prices:

```
function _calculateDeviationBps(uint160 price1, uint160 price2) internal pure returns
(uint256) {
    uint256 larger;
    uint256 smaller;

    if (price1 > price2) {
        larger = uint256(price1);
        smaller = uint256(price2);
    } else {
        larger = uint256(price2);
        smaller = uint256(price1);
    }

    return FullMath.mulDiv(larger - smaller, BPS_DENOMINATOR, larger);
}
```

Since `sqrtPriceX96` represents the square root of the price, a deviation limit applied to this value allows a larger movement in actual price terms. For example, with the default `maxDeviationBps` =

200, a **2%** movement in square root price corresponds to approximately a **3.96%** movement in the underlying price.

As a result, TWAP-backed orders can pass the spot-vs-TWAP deviation check at a wider price movement than intended.

Remediation:

- Compute spot-vs-TWAP deviation in actual price space, not square root price space.
- Convert both **sqrtPriceX96** values to comparable price values before applying **maxDeviationBps**.
- Apply the same corrected deviation logic in all TWAP validation paths.

ALIEN-37 | TWAP oracle source can change after route execution

Acknowledged

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

contracts/EpsilonRouter.sol#L313

Description:

EpsilonRouter checks order readiness before executing the swap route, but output validation performs oracle selection again after the route has already executed. The TWAP pool used for readiness is not snapshotted or bound to the later validation step.

```
uint256 currentPrice = OracleLib.getPriceWithFallback(
    priceVerifier,
    fallbackOracles,
    order.tokenIn,
    order.tokenOut
);
if (!_isOrderReady(order, currentPrice)) revert OrderNotReady();

amountOut = _executeSwapRoutes(order, routes, makingAmount);

OracleLib.validateSwapOutput(
    priceVerifier,
    order.tokenIn,
    order.tokenOut,
    makingAmount,
    amountOut,
    _effectiveSlippage(order),
    order.makerTraits.isStopLoss(),
    skipValidation
);
```

During validation, the verifier reselects the oracle. If no Redstone or TWAP source is available at that later point, validation returns without enforcing an output floor.

```
(bool useRedstone, bool useTWAP) = _selectOracle(tokenIn, tokenOut);

if (!useRedstone && !useTWAP) {
    return;
}
```

TWAP availability depends on **getBestPool()**, which dynamically checks Uniswap V3 active liquidity and observation cardinality. Since active liquidity can change during a route execution, the pool selected before the route may fail the gate afterward. Validation may then either skip entirely if no pool remains, or switch to a lower-priority pool with a less favorable TWAP.

```
for (uint256 i = 0; i < feeTierPreference.length; i++) {
    pool = factory.getPool(token0, token1, feeTierPreference[i]);
    if (pool != address(0)) {
        IUniswapV3Pool poolContract = IUniswapV3Pool(pool);
        (, , uint16 observationCardinality, , , ) = poolContract.slot0();
        uint128 liquidity = poolContract.liquidity();

        if (observationCardinality >= minObservationCardinality && liquidity >= minLiquidity) {
            return pool;
        }
    }
}

return address(0);
```

For TWAP-only pairs, this allows a route to affect the preferred pool before validation runs. The maker's output may then be checked against no TWAP source, or against a different lower-priority pool than the one that made the order executable.

Remediation:

- Bind output validation to the same oracle source used during readiness evaluation.
- Snapshot the selected TWAP pool before route execution and require validation to use that pool.
- Revert if the snapshotted TWAP source is no longer valid during post-route validation.
- Do not silently return from validation for pairs that were oracle-validated during readiness.
- Avoid falling back to a different TWAP pool during the same execution unless the order explicitly allows that behavior.

ALIEN-22 | Oracle slippage check uses gross input instead of fee-adjusted amount

Fixed 

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

`-/contracts/EpsilonRouter.sol`

Description:

When `order.makerTraits.takeFeesInTokenIn()` is true, `_executeSwapRoutes` internally deducts protocol, keeper, and referral fees from `makingAmount` before routing, only `amountToSwap = makingAmount - fees` reaches the DEX. The oracle validation call that follows, however, uses the gross `makingAmount` as the input reference:

```
amountOut = _executeSwapRoutes(order, routes, makingAmount); // DEX gets amountToSwap
```

```
OracleLib.validateSwapOutput(  
    priceVerifier, order.tokenIn, order.tokenOut,  
    makingAmount,  
    amountOut, _effectiveSlippage(order), ...  
);
```

The oracle computes `expectedOut = makingAmount * price` and enforces `amountOut >= expectedOut * (1 - maxSlippage)`. Because `amountOut` was produced by swapping only `amountToSwap` (fee-reduced), the implied rate `amountOut / makingAmount` will look like slippage to the oracle.

When `order.makerTraits.takeFeesInTokenIn()` is enabled, `_executeSwapRoutes` deducts protocol, keeper, and referral fees from `makingAmount` before performing the swap. This means the DEX only receives `amountToSwap = makingAmount - fees`, not the full `makingAmount`.

However, after the swap, the oracle validation still uses the original `makingAmount` as the reference input:

```
amountOut = _executeSwapRoutes(order, routes, makingAmount); // DEX gets amountToSwap
```

```
OracleLib.validateSwapOutput(  
    priceVerifier, order.tokenIn, order.tokenOut,  
    makingAmount,
```

```
amountOut, _effectiveSlippage(order), ...  
);
```

The oracle calculates the expected output based on **makingAmount * price** and checks that **amountOut** is within the allowed slippage range. Since **amountOut** was produced using a smaller input (**amountToSwap**), the resulting rate appears lower when compared against **makingAmount**. This makes the swap look like it incurred slippage, even if execution was correct, because the oracle is effectively comparing net output against a gross input.

Remediation:

Compute **amountToSwap** (amount without the fees) before **_executeSwapRoutes** and pass it to **validateSwapOutput** instead of **makingAmount**, as done as in **executeTrusted**.

ALIEN-27 | FillOrder Accepts all Order Types

Fixed 

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Description:

fillOrder has no guard against **allowMultipleFills** orders. Once **registerDca** registers a DCA order, any **globalKeeper** can call **fillOrder** on the same order. **executeDcaOrder** never updates the remaining invalidator, it tracks execution state exclusively in **data.slot2**. **fillOrder** therefore sees the remaining invalidator as fully unspent (**order.amountIn**). The DCA accounting is unaffected: **executeDcaOrder** can still complete all intended slices afterwards, resulting in **totalExecutions + x** total actions.

Remediation:

fillOrder should enforce that the order is not a DCA or trailing/stop-loss type, ensuring only standard orders can be filled through this function.

For example:

```
if (order.makerTraits.allowMultipleFills()) revert NotAllowed(); // DCA
if (order.makerTraits.isTrailing()) revert NotAllowed(); // trailing stop
if (order.makerTraits.isStopLoss() || order.triggerPrice != 0) revert NotAllowed(); // stop-loss / take-profit
```


ALIEN-35 | Lack of slippage checks for makers in `matchOrders` and `batchMatchOrders` functions

Acknowledged

Severity:

Medium

Probability:

Likely

Impact:

Medium

Path:

`contracts/EpsilonMatcher.sol#L135-L164`

`contracts/EpsilonMatcher.sol#L181-L212`

Description:

The `matcherPullLeg()` function in the `EpsilonRouter` performs only one price-related check on each order by triggering `_isOrderReady(order, oraclePrice)`. If it passes, the amount received is calculated by applying different fees to the oracle price, specifically the keeper fees. The order's `maxSlippageBps` is not consulted in `matcherPullLeg()` at all.

The `matchOrders()` function in the `EpsilonMatcher` contract matches sell and buy orders by calling `matcherPullLeg()` on the router for both orders. However, this function does not run any slippage checks for the makers of either order, even though the keeper fees are passed directly by the keeper. Therefore, the maker will effectively pay the full total of `matchingFeeBps + keeperFeeBps + referralFeeBps` without being ensured of the slippage tolerance they signed (`maxSlippageBps`).

In the matcher contract, `matchRingOrders` and `matchRingOrdersHybrid` already perform slippage checks for makers by running an explicit per-leg fairness check in their own loop (using `outputAmounts[i] * (BPS - maxSlippage) / BPS`). However, `matchOrders` and `batchMatchOrders` lack this validation.

```
function matchOrders(
    Order calldata sellOrder,
    bytes calldata sellSig,
    Order calldata buyOrder,
    bytes calldata buySig,
    uint256 matchAmountA,
    uint16 keeperFeeBps
) external onlyKeeper returns (uint256 amountB) {
    if (matchAmountA == 0) revert SwapWithZeroAmount();
    if (sellOrder.tokenIn != buyOrder.tokenOut || sellOrder.tokenOut != buyOrder.tokenIn) revert
    OrdersNotOpposite();

    bytes32 sellHash = ROUTER.hashOrder(sellOrder);
```

```

bytes32 buyHash = ROUTER.hashOrder(buyOrder);

address buyRecv = buyOrder.receiver != address(0) ? buyOrder.receiver : buyOrder.maker;
address sellRecv = sellOrder.receiver != address(0) ? sellOrder.receiver : sellOrder.maker;

// Leg 1: pull tokenA from sell.maker, deliver net to buyer
(, uint256 oraclePrice) = _callMatcherPullLeg(sellOrder, sellSig, matchAmountA, buyRecv, msg.sender,
keeperFeeBps);

// Compute amount of tokenB owed to seller at oracle clearing price
uint256 sellDec = 10 ** IERC20Metadata(sellOrder.tokenIn).decimals();
amountB = (matchAmountA * oraclePrice) / sellDec;
if (amountB == 0) revert InvalidAmount();

// Leg 2: pull tokenB from buy.maker, deliver net to seller
_callMatcherPullLeg(buyOrder, buySig, amountB, sellRecv, msg.sender, keeperFeeBps);

emit OrdersMatched(sellHash, buyHash, msg.sender, matchAmountA, amountB, oraclePrice);
}

```

```

function _batchMatchInner(
    Order calldata sellOrder,
    bytes calldata sellSig,
    Order[] calldata buyOrders,
    bytes[] calldata buySigs,
    uint256[] calldata matchAmounts,
    uint16 keeperFeeBps
) private {
    bytes32 sellHash = ROUTER.hashOrder(sellOrder);
    address sellRecv = sellOrder.receiver != address(0) ? sellOrder.receiver : sellOrder.maker;
    uint256 sellDec = 10 ** IERC20Metadata(sellOrder.tokenIn).decimals();

    for (uint256 i; i < buyOrders.length;) {
        uint256 amtA = matchAmounts[i];
        if (amtA == 0) revert SwapWithZeroAmount();

        Order calldata buy = buyOrders[i];
        if (sellOrder.tokenIn != buy.tokenOut || sellOrder.tokenOut != buy.tokenIn) revert OrdersNotOpposite();

        address buyRecv = buy.receiver != address(0) ? buy.receiver : buy.maker;

        (, uint256 oraclePrice) = _callMatcherPullLeg(sellOrder, sellSig, amtA, buyRecv, msg.sender,
keeperFeeBps);
    }
}

```

```

uint256 amtB = (amtA * oraclePrice) / sellDec;
if (amtB == 0) revert InvalidAmount();

_callMatcherPullLeg(buy, buySigs[i], amtB, sellRecv, msg.sender, keeperFeeBps);

emit OrdersMatched(sellHash, ROUTER.hashOrder(buy), msg.sender, amtA, amtB, oraclePrice);
unchecked { ++i; }
}
}

```

Remediation:

Consider adding slippage validations for makers to `matchOrders` and `_batchMatchInner` function, mirroring the loop logic used in `matchRingOrders`.

ALIEN-10 | Fallback oracle prices are used without consistent validation

Acknowledged

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

contracts/libraries/OracleLib.sol#L29

Description:

`OracleLib.getPriceWithFallback()` accepts a fallback adapter price when the primary verifier returns zero or fails softly.

```
price = _tryRedstonePrice(primaryVerifier, tokenIn, tokenOut);
if (price != 0) {
    if (price > type(uint128).max) revert PriceTooHigh();
    return price;
}
price = _tryFallbackAdapters(fallbackAdapters, tokenIn, tokenOut);
if (price != 0) return price;
```

However, swap output validation is always routed through the primary verifier. The fallback adapter used for readiness is not passed into the validation step.

```
IEpsilonPriceVerifier(primaryVerifier).setSwapValidationContext(
    tokenIn, tokenOut, amountIn, amountOut, maxSlippageOverride, isStopLoss
);

_forwardRedstoneCallStrict(VALIDATE_SWAP_OUTPUT_SELECTOR, primaryVerifier);
```

If the primary verifier has no usable oracle for the pair, validation returns without checking the received output.

```
function validateSwapOutput() external view {
    // Full bypass for trusted keeper
    if (pendingSkipValidation) {
        return;
    }
}
```

```

address tokenIn = pendingTokenIn;
address tokenOut = pendingTokenOut;
uint256 amountIn = pendingAmountIn;
uint256 amountOut = pendingAmountOut;
uint256 maxSlippageOverride = pendingMaxSlippageOverride;
bool isStopLoss = pendingIsStopLoss;

(bool useRedstone, bool useTWAP) = _selectOracle(tokenIn, tokenOut);

// No oracle configured - skip validation
if (!useRedstone && !useTWAP) {
    return;
}

uint256 maxSlippage = maxSlippageOverride > 0 ? maxSlippageOverride : maxSlippageBps;
if (isStopLoss) maxSlippage = maxSlippage * 2;

if (useTWAP) {
    // Use TWAP oracle for validation
    twapOracle.validateSwapOutput(tokenIn, tokenOut, amountIn, amountOut, maxSlippage);
} else {
    // Use Redstone for validation
    _validateWithRedstone(tokenIn, tokenOut, amountIn, amountOut, maxSlippage);
}
}

```

This creates inconsistent behavior: fallback pricing can make an order executable, while the corresponding output validation may be skipped or performed against a different oracle source. The same fallback price is also used directly as the clearing price in matcher settlement paths. In **matcherPullLeg**, the router obtains **oraclePrice** from **getPriceWithFallback()** and returns it to the matcher.

```

function matcherPullLeg(
    Order calldata order,
    bytes calldata signature,
    uint256 fillAmount,
    address recipient,
    address keeperRecipient,
    uint16 keeperFeeBps
) external nonReentrant whenNotPaused onlyAuthorizedMatcher returns (uint256 netToRecipient, uint256 oraclePrice) {
    if (keeperFeeBps > maxKeeperFeeBps) revert FeeTooHigh();
}

```

```

    _currentKeeperFeeBps = keeperFeeBps;
    if (fillAmount == 0) revert SwapWithZeroAmount();
    bytes32 orderHash = hashOrder(order);
    _validateMatchSig(order.maker, orderHash, signature);
    if (order.makerTraits.isExpired()) revert OrderExpired();

    uint256 remaining = _checkRemainingMakingAmount(order, orderHash);
    if (fillAmount > remaining) revert MatchAmountTooLarge();

    oraclePrice = OracleLib.getPriceWithFallback(priceVerifier, fallbackOracles, order.tokenIn,
order.tokenOut);
    if (oraclePrice == 0) revert OrderNotReady();
    if (order.triggerPrice != 0 && !_isOrderReady(order, oraclePrice)) revert OrderNotReady();

    _invalidateOrder(order, orderHash, remaining, fillAmount, order.maker);
    netToRecipient = _matcherDistribute(order.tokenIn, order.maker, recipient, fillAmount, order.referrer,
order.referralFeeBps, keeperRecipient);
}

```

The matcher then uses that value to compute the counter-leg amount, without a later swap output validation phase for pure P2P/ring settlement.

```

function matchOrders(
    Order calldata sellOrder,
    bytes calldata sellSig,
    Order calldata buyOrder,
    bytes calldata buySig,
    uint256 matchAmountA,
    uint16 keeperFeeBps
) external onlyKeeper returns (uint256 amountB) {
    if (matchAmountA == 0) revert SwapWithZeroAmount();
    if (sellOrder.tokenIn != buyOrder.tokenOut || sellOrder.tokenOut != buyOrder.tokenIn) revert
OrdersNotOpposite();

    bytes32 sellHash = ROUTER.hashOrder(sellOrder);
    bytes32 buyHash = ROUTER.hashOrder(buyOrder);

    address buyRecv = buyOrder.receiver != address(0) ? buyOrder.receiver : buyOrder.maker;
    address sellRecv = sellOrder.receiver != address(0) ? sellOrder.receiver : sellOrder.maker;

    // Leg 1: pull tokenA from sell.maker, deliver net to buyer
    (, uint256 oraclePrice) = _callMatcherPullLeg(sellOrder, sellSig, matchAmountA, buyRecv, msg.sender,
keeperFeeBps);
}

```

```

// Compute amount of tokenB owed to seller at oracle clearing price
uint256 sellDec = 10 ** IERC20Metadata(sellOrder.tokenIn).decimals();
amountB = (matchAmountA * oraclePrice) / sellDec;
if (amountB == 0) revert InvalidAmount();

// Leg 2: pull tokenB from buy.maker, deliver net to seller
_callMatcherPullLeg(buyOrder, buySig, amountB, sellRecv, msg.sender, keeperFeeBps);

emit OrdersMatched(sellHash, buyHash, msg.sender, matchAmountA, amountB, oraclePrice);
}

```

As a result, when fallback adapters are configured and their prices diverge from the intended primary source, orders may execute or settle using fallback prices without equivalent validation.

Remediation:

- Use the same oracle source for readiness, settlement pricing, and output validation.
- If a fallback adapter supplies the execution price, validate swap output against that fallback price as well.
- Do not allow matcher settlement to use fallback prices unless the fallback source has passed equivalent validation.
- Treat primary verifier failure as a hard failure for paths that require primary-priced execution.

ALIEN-2 | Missing Order Parameter Validation in executeDcaOrder function

Fixed 

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

contracts/EpsilonRouter.sol#L337-L357

Description:

In `EpsilonRouter.executeDcaOrder()`, the function allows a caller to register a new order using the maker's signature if the order does not already exist (lines 337–357):

```
if (data.slot2 == 0 && !data.exists) {
    _verifySignature(maker, orderHash, signature);
    if (!order.makerTraits.allowMultipleFills()) revert NotDcaOrder();
    if (params.intervalSeconds == 0 || params.totalExecutions == 0) revert InvalidParams();
    if (params.flexibilityBps > 2000) revert InvalidParams();

    uint256 baseAmount;
    unchecked {
        baseAmount = params.totalAmountIn / params.totalExecutions;
    }
    if (baseAmount != order.amountIn) revert DcaAmountMismatch();

    unchecked {
        data.slot1 = (uint256(params.totalExecutions) << 128) | uint256(params.intervalSeconds);
    }
    data.exists = true;
    _orderCreator[orderHash] = maker;
    data.totalAmountIn = params.totalAmountIn;
    data.flexibilityBps = params.flexibilityBps;
    data.shortCircuit = params.shortCircuit;
}
```

However, unlike `registerDcaOrder()`, this flow does not call `_validateOrderParams()`.


```
function registerDcaOrder(  
    Order calldata order,  
    uint128 intervalSeconds,  
    uint128 totalExecutions,  
    uint128 totalAmountIn,  
    uint16 flexibilityBps,  
    bool shortCircuit,  
    address _designatedKeeper  
) external returns (bytes32 orderHash) {  
    _validateOrderParams(order);
```

As a result, an order can be created with invalid or inconsistent parameters when registered through **executeDcaOrder()**.

Furthermore, the issue also happens in functions **matcherPullLeg()**.

Remediation:

Consider adding the **_validateOrderParams** after line 337.

ALIEN-13 | executeTrusted() uses stale _currentKeeperFeeBps from previous action of Router

Fixed ✓

Severity:

Medium

Probability:

Likely

Impact:

Medium

Path:

contracts/EpsilonRouter.sol#L503-L526

Description:

`executeTrusted()` reads `_currentKeeperFeeBps` but never sets it. Unlike every other fill function (`fillOrder`, `executeDcaOrder`, `executeTrailingStop`, `matcherPullLeg`, `matcherExecuteHybridLeg`), `executeTrusted` does not accept a `keeperFeeBps` parameter and does not initialize `_currentKeeperFeeBps`.

Since `_currentKeeperFeeBps` is a regular storage slot (uint16 private) that persists across transactions, `executeTrusted` therefore operates on whatever value was last written by any prior action of router. As a result, the fee in `executeTrusted` function may be calculated by the stale `_currentKeeperFeeBps`.

```
uint256 amountToSwap;  
unchecked {  
    amountToSwap = order.makerTraits.takeFeesInTokenIn()  
        ? makingAmount - ((makingAmount * (protocolFeeBps + _currentKeeperFeeBps + order.referralFeeBps))  
        / BPS_DENOMINATOR)  
        : makingAmount;  
}
```

The maker risks losing 5% (`maxKeeperFeeBps`) as fee, even when the keeper does not intend to collect it.

```
function executeTrusted(Order calldata order, bytes calldata signature, address[] calldata targets, bytes[]  
calldata callDatas) external nonReentrant whenNotPaused returns (uint256 amountOut) {  
    if (!trustedKeepers[msg.sender]) revert InvalidKeeper();  
    if (!order.makerTraits.trustKeeper()) revert TrustedExecutionNotAllowed();  
    if (targets.length != callDatas.length) revert LengthMismatch();  
  
    bytes32 orderHash = hashOrder(order);  
    address maker = order.maker;
```

```

_validateOrderParams(order);
_validateMatchSig(maker, orderHash, signature);
_checkReentrancy(order, orderHash, maker);
if (order.makerTraits.isExpired()) revert OrderExpired();

uint256 makingAmount = order.makerTraits.useBitInvalidator()
    ? order.amountIn
    : _checkRemainingMakingAmount(order, orderHash);
if (makingAmount == 0) revert SwapWithZeroAmount();

uint256 amountToSwap;
unchecked {
    amountToSwap = order.makerTraits.takeFeesInTokenIn()
        ? makingAmount - ((makingAmount * (protocolFeeBps + _currentKeeperFeeBps +
order.referralFeeBps)) / BPS_DENOMINATOR)
        : makingAmount;
}

```

Remediation:

Add a **keeperFeeBps** parameter to **executeTrusted** and initialize **_currentKeeperFeeBps** in **executeTrusted**:

```

function executeTrusted(
    Order calldata order, bytes calldata signature,
    address[] calldata targets, bytes[] calldata callDatas,
    + uint16 keeperFeeBps
) external nonReentrant whenNotPaused returns (uint256 amountOut) {
    + if (keeperFeeBps > maxKeeperFeeBps) revert FeeTooHigh();
    + _currentKeeperFeeBps = keeperFeeBps;
    // ... rest of function
}

```

ALIEN-8 | Order Type Should Be Included in Order Struct

Acknowledged

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

contracts/EpsilonRouter.sol#L736-L745

Description:

The current **Order** struct is defined as:

```
struct Order {  
    uint256 salt;  
    address maker;  
    address receiver;  
    address tokenIn;  
    address tokenOut;  
    uint256 amountIn;  
    uint256 triggerPrice;  
    MakerTraits makerTraits;  
    address referrer;  
    uint16 referralFeeBps;  
}
```

It does not include any field indicating the order type. Additionally, **_verifySignature()** only validates the signature against the order hash, without considering whether the order is a DCA, trailing stop, or normal order.

As a result, an attacker can submit the same signed order under a different order type.

For example, if a user intends to create a DCA order, an attacker can front-run and register it as a normal order instead (via **registerOrder**, which is permissionless, or **fillOrder**, which requires a keeper). This causes the order to be executed under a different logic than intended by the user.

```
function _verifySignature(address maker, bytes32 orderHash, bytes calldata signature) internal view {  
    if (signature.length == 65) {  
        bytes32 r; bytes32 s; uint8 v;  
        assembly { r := calldataload(signature.offset) s := calldataload(add(signature.offset, 0x20)) v := byte(0,  
calldataload(add(signature.offset, 0x40))) }  
        address signer = ECDSA.recover(orderHash, v, r, s);  
        if (signer == maker && signer != address(0)) return;  
    }  
}
```

```

    }
    try IERC1271(maker).isValidSignature(orderHash, signature) returns (bytes4 magicValue) { if (magicValue
== ERC1271_MAGIC_VALUE) return; } catch {}
    revert BadSignature();
}

```

Remediation:

Include the order type in the Order struct so it is part of the signed data:

```

++ enum OrderType {
++     Normal,
++     DCA,
++     TrailingStop
++ }

struct Order {
    uint256 salt;
    address maker;
    address receiver;
    address tokenIn;
    address tokenOut;
    uint256 amountIn;
    uint256 triggerPrice;
    MakerTraits makerTraits;
    address referrer;
    uint16 referralFeeBps;
++     OrderType orderType;
}

```

Additionally, the functions **fillOrder()**, **executeDcaOrder()**, **executeTrailingStop()** should validate the order's type before executing the order.

ALIEN-12 | Redstone payload freshness check uses inconsistent timestamp units

Fixed 

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

contracts/EpsilonPriceVerifier.sol

Description:

EpsilonPriceVerifier extracts Redstone values and timestamps with `getOracleNumericValuesAndTimestampFromTxMsg()` and then performs a local freshness check.

```
(uint256[] memory values, uint256 timestamp) =  
    getOracleNumericValuesAndTimestampFromTxMsg(feedIds);  
  
if (block.timestamp > timestamp + maxPriceStaleness) revert PriceStale();
```

Redstone payload timestamps are expressed in milliseconds. The Redstone default validator converts the received timestamp to seconds before comparing it with `block.timestamp`.

```
function validateTimestamp(uint256 receivedTimestampMilliseconds) internal view {  
    // Getting data timestamp from future seems quite unlikely  
    // But we've already spent too much time with different cases  
    // Where block.timestamp was less than dataPackage.timestamp.  
    // Some blockchains may case this problem as well.  
    // That's why we add MAX_BLOCK_TIMESTAMP_DELAY  
    // and allow data "from future" but with a small delay  
    uint256 receivedTimestampSeconds = receivedTimestampMilliseconds / 1000;  
  
    if (block.timestamp < receivedTimestampSeconds) {  
        if ((receivedTimestampSeconds - block.timestamp) >  
            DEFAULT_MAX_DATA_TIMESTAMP_AHEAD_SECONDS) {  
            revert TimestampFromTooLongFuture(receivedTimestampSeconds, block.timestamp);  
        }  
    } else if ((block.timestamp - receivedTimestampSeconds) >  
        DEFAULT_MAX_DATA_TIMESTAMP_DELAY_SECONDS) {  
        revert TimestampsTooOld(receivedTimestampSeconds, block.timestamp);  
    }  
}
```

```
}  
}
```

However, `getOracleNumericValuesAndTimestampFromTxMsg()` returns the timestamp without applying Redstone's timestamp validation.

```
function getOracleNumericValuesAndTimestampFromTxMsg(bytes32[] memory dataFeedIds)  
    internal  
    view  
    virtual  
    returns (uint256[] memory, uint256)  
{  
    return _securelyExtractOracleValuesAndTimestampFromTxMsg(dataFeedIds);  
}
```

As a result, **EpsilonPriceVerifier** compares a millisecond Redstone timestamp directly against the second-based `block.timestamp`. This makes **maxPriceStaleness** ineffective on Redstone-backed price reads, swap output validation, token USD price reads, and trigger checks that call `_getRedstonePriceInternal()`.

If an old signed Redstone payload is available and its relative price is favorable, it can be reused in a later router execution and treated as fresh.

Remediation:

- Normalize Redstone timestamps to seconds before comparing them with `block.timestamp`.
- Prefer Redstone's timestamp validation helper where Redstone payload timestamps are consumed.
- Apply the corrected freshness check consistently across all Redstone-backed price consumers.

ALIEN-23 | Surplus Taken Before Fees Causes Inverted Incentives

Fixed 

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

`./contracts/EpsilonRouter.sol`

Description:

During `_finalizeFill` it currently removes surplus first and apply fees afterwards, however a surplus should be calculated from the net amount out, after all fees are deducted.

Example

- Input: 100 USDC
- Expected (oracle): 100 USDT
- Actual Out: 110 USDT
- Fee: 10%
- Surplus: 10% → split 5 / 5

Without surplus:

Amount Out = $110 \times 0.9 = 99$

With surplus:

Adjusted = $110 - 5 = 105$

Amount Out = $105 \times 0.9 = 94.5$ (should be 99)

The surplus accounting is inverted: the user is penalized for better execution rather than benefiting from it.

Remediation:

Apply fees first, then capture the surplus.

ALIEN-32 | Incorrect Permit2 interface will causes permit2 to be permanently broken

Fixed 

Severity:

Medium

Probability:

Unlikely

Impact:

Medium

Path:

contracts/EpsilonRouter.sol:IPermit2#L19-L23

Description:

The EpsilonRouter contract declares the IPermit2 interface at the beginning of the contract. This is to be used in the **swap** function to allow users to use Uniswap's permit2 to transfer tokens.

However, the interface that is declared uses an incorrect **PermitTransferFrom** struct compared to the real Uniswap's permit2 contract.

In EpsilonRouter:

```
struct PermitTransferFrom { address token; uint256 amount; }
```

And in Uniswap:

```
struct TokenPermissions {  
    address token;  
    uint256 amount;  
}  
  
struct PermitTransferFrom {  
    TokenPermissions permitted;  
    uint256 nonce;  
    uint256 deadline;  
}
```

This means that permit2 usage in **swap** will always cause a revert due to ABI-decoding at permit2.

```
interface IPermit2 {  
    struct PermitTransferFrom { address token; uint256 amount; }  
    struct SignatureTransferDetails { address to; uint256 requestedAmount; }  
    function permitTransferFrom(PermitTransferFrom calldata permit, SignatureTransferDetails calldata
```

```
transferDetails, address owner, bytes calldata signature) external;  
}
```

Remediation:

Use the correct interface in EpsilonRouter.

ALIEN-31 | Order cancellation does not validate owner on Indexer side

Fixed 

Severity:

Medium

Probability:

Likely

Impact:

Medium

Path:

indexer-go/cmd/indexer/indexer.go:handleOrderCancelled#L671-L684

Description:

The function **cancelOrder()** on the EpsilonRouter.sol contracts allows a caller to invalidate/cancel an order hash. The logic in the contract is correct where the order hash is invalidated for a mapping of **msg.sender** and the order is only deleted if **msg.sender** is the order creator:

```
else {
    _remainingInvalidator[msg.sender][orderHash] = RemainingInvalidatorLib.fullyFilled();
    emit OrderCancelled(orderHash);
}
if (_orderCreator[orderHash] == msg.sender) {
    delete orderData[orderHash];
    delete designatedKeeper[orderHash];
    delete orderPostHook[orderHash];
    delete _orderCreator[orderHash];
}
```

The contract will still emit the **OrderCancelled(orderHash)** event regardless and this only contains the order hash information.

As a result, the OrderCancelled event log processor on the Indexer side does not validate whether the caller was the order creator and unconditionally updates the database to set the order status as cancelled:

```
hash := l.Topics[1].Hex()
if _, err := idx.db.ExecContext(cctx, "UPDATE orders SET
status='cancelled',updated_at=CURRENT_TIMESTAMP WHERE order_hash=?", hash); err != nil {
    return fmt.Errorf("cancel: %w", err)
}
```

This means that anyone can call **EpsilonRouter.sol:cancelOrder** to cancel any arbitrary order hash on the Indexer's side, making it so that the keeper will never execute the order.

```
func (idx *Indexer) handleOrderCancelled(ctx context.Context, l types.Log) error {
    eventsProcessed.WithLabelValues("cancelled").Inc()
    if len(l.Topics) < 2 {
        return nil
    }
    cctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()
    hash := l.Topics[1].Hex()
    if _, err := idx.db.ExecContext(cctx, "UPDATE orders SET
status='cancelled',updated_at=CURRENT_TIMESTAMP WHERE order_hash=?", hash); err != nil {
        return fmt.Errorf("cancel: %w", err)
    }
    idx.logger.Info("OrderCancelled", "hash", safeHash(hash))
    return nil
}
```

Remediation:

Consider whether the **OrderCancelled** event should only be emitted in the second branch, where **msg.sender** is the order creator and the order is actually fully deleted.

ALIEN-33 | TWAP price limits the tick price to 128-bit

Fixed 

Severity:

Low

Probability:

Rare

Impact:

Medium

Path:

contracts/EpsilonTWAPOracle.sol:getTWAPPrice#L171-L192

Description:

The function **getTWAPPrice** is used in the EpsilonPriceVerifier to get the pool's current TWAP price from the TWAP SqrtPriceX96 using **pool.observe**.

In this function it squares the tick price by multiplying it by itself:

```
uint256 priceX192 = uint256(sqrtPriceX96TWAP) * uint256(sqrtPriceX96TWAP);
```

However, by taking the square of the price, the function will revert with an overflow if the value is greater than or equal to 128-bits, resulting in 256-bits or more.

The SqrtPriceX96 is a **uint160** and a 64X96 precision number, which means that the 64 most significant bits are used for the whole number, and the 96 least significant bits are for the decimals. This also means that the SqrtPriceX96 is always 96+ bits by default.

By limiting support for the SqrtPriceX96 to only 128-bits, it means that there are only 32 of the most significant bits left for the whole number, instead of 64 bits. In other words, the square root price ratio between token A and token B is limited to 4.29 billion (32-bit), instead of a much higher value with 64 bits.

While tokens with decently high USD value might not encounter these price ratios, it is still definitely possible if a highly valued token with a low amount of decimals (e.g. WBTC) is paired with a token with 18 decimals and a low value.

In such a case, the **getTWAPPrice** function would cause a revert.

```
function getTWAPPrice(address tokenIn, address tokenOut) external view returns (uint256 price) {
    address pool = getBestPool(tokenIn, tokenOut);
    if (pool == address(0)) revert PoolNotFound();

    uint160 sqrtPriceX96TWAP = _getSqrtPriceX96TWAP(IUniswapV3Pool(pool));

    uint256 priceX192 = uint256(sqrtPriceX96TWAP) * uint256(sqrtPriceX96TWAP);
```

```

(address token0, ) = tokenIn < tokenOut ? (tokenIn, tokenOut) : (tokenOut, tokenIn);

uint8 decimalsIn = IERC20Metadata(tokenIn).decimals();

if (tokenIn == token0) {
    // Pool stores token1/token0, multiply by 10^decimalsIn to convert from wei ratio
    price = FullMath.mulDiv(priceX192, 10 ** decimalsIn, uint256(1) << 192);
} else {
    // Need to invert: pool stores token1/token0 but we want token0/token1
    price = FullMath.mulDiv(uint256(1) << 192, 10 ** decimalsIn, priceX192);
}

if (price == 0) revert InvalidPrice();
}

```

Remediation:

Perform a different calculation for the final price if the **sqrtPriceX96TWAP** is greater than 128 bits, such that it will not revert from overflow.

ALIEN-30 | Inconsistent zero-slippage handling in ring matcher paths

Acknowledged

Severity:

Low

Probability:

Unlikely

Impact:

Low

Path:

contracts/EpsilonMatcher.sol#L257

Description:

The router and matcher use different interpretations for `maxSlippageBps == 0`. In the router, a zero value means that the order should use the configured pair default, or the global default if no pair-specific value is set.

```
function _effectiveSlippage(Order calldata order) internal view returns (uint256) {  
    uint256 makerBps = order.makerTraits.maxSlippageBps();  
    if (makerBps != 0) return makerBps;  
    uint16 pairBps = pairDefaultSlippageBps[_pairKey(order.tokenIn, order.tokenOut)];  
    if (pairBps != 0) return pairBps;  
    return globalDefaultSlippageBps;  
}
```

In the ring and hybrid matcher paths, the same zero value is handled differently. Instead of applying the router's default slippage policy, the matcher uses the sum of matching, keeper, and referral fees as the fallback threshold.

```
uint256 totalFeeBps = _matchingFeeBps + _keeperFeeBps + orders[next].referralFeeBps;  
...  
uint256 maxSlippage = orders[i].makerTraits.maxSlippageBps();  
if (maxSlippage == 0) maxSlippage = totalFeeBps;  
...  
if (netReceived < minAcceptable) revert RingPriceUnfair();
```

```
uint256 totalFeeBps = _matchingFeeBps + _keeperFeeBps + orders[next].referralFeeBps;  
...  
uint256 maxSlippage = orders[i].makerTraits.maxSlippageBps();  
if (maxSlippage == 0) maxSlippage = totalFeeBps;  
...  
if (netReceived < minAcceptable) revert RingPriceUnfair();
```

This makes the effective slippage policy depend on the execution path. A maker who leaves **maxSlippageBps** unset may be protected by pair/global defaults in normal router execution, but by fee-derived thresholds in ring or hybrid matching. This can lead to different acceptance behavior for orders that otherwise carry the same slippage setting.

```
(address token0, ) = tokenIn < tokenOut ? (tokenIn, tokenOut) : (tokenOut, tokenIn);

uint8 decimalsIn = IERC20Metadata(tokenIn).decimals();

if (tokenIn == token0) {
    // Pool stores token1/token0, multiply by 10^decimalsIn to convert from wei ratio
    price = FullMath.mulDiv(priceX192, 10 ** decimalsIn, uint256(1) << 192);
} else {
    // Need to invert: pool stores token1/token0 but we want token0/token1
    price = FullMath.mulDiv(uint256(1) << 192, 10 ** decimalsIn, priceX192);
}

if (price == 0) revert InvalidPrice();
}
```

Remediation:

Apply a single zero-slippage fallback policy across router, ring matcher, and hybrid matcher execution paths.

ALIEN-29 | Hybrid overflow keeper fees are sent to the matcher contract

Fixed 

Severity:

Low

Probability:

Unlikely

Impact:

Low

Path:

contracts/EpsilonRouter.sol#L893

Description:

In hybrid ring matching, the matcher passes the actual keeper as **keeperRecipient** when calling the router.

```
(, , uint256 oraclePrice) = _callMatcherExecuteHybridLeg(
    orders[i],
    p.signatures[i],
    p.ringAmounts[i],
    overflowAmount,
    prevReceiver,
    msg.sender,
    p.overflows[i].routes,
    keeperFeeBps
);
```

The ring portion uses this value correctly when distributing keeper fees.

```
ringNet = _matcherDistribute(
    order.tokenIn,
    order.maker,
    recipient,
    ringAmount,
    order.referrer,
    order.referralFeeBps,
    keeperRecipient
);
```

```
if (kFee > 0) token.safeTransfer(keeperRecipient, kFee);
```

However, the overflow portion is finalized through the generic fill path.

```
if (overflowAmount > 0) {
    overflowOut = _executeSwapRoutes(order, overflowRoutes, overflowAmount);
    bool skipValidation = _shouldSkipValidation(order, orderHash);
    OracleLib.validateSwapOutput(priceVerifier, order.tokenIn, order.tokenOut, overflowAmount, overflowOut,
    _effectiveSlippage(order), order.makerTraits.isStopLoss(), skipValidation);
    _finalizeFill(order, orderHash, order.maker, overflowAmount, overflowOut, oraclePrice);
    emit OrderFilled(orderHash, order.maker, keeperRecipient, overflowAmount, overflowOut, 0);
}
```

That path sends keeper fees to `msg.sender`.

```
if (protocolFee > 0) feeToken.safeTransfer(feeCollector, protocolFee);
if (keeperFee > 0) feeToken.safeTransfer(msg.sender, keeperFee);
```

During a hybrid matcher call, `msg.sender` in the router is the **EpsilonMatcher** contract, not the keeper that initiated the match. As a result, keeper fees from the overflow portion are transferred to the matcher contract instead of the keeper. Since the matcher does not expose an ERC20 withdrawal path, these fee tokens can remain in the matcher contract.

Remediation:

Pass the explicit keeper recipient through the hybrid overflow finalization path.

ALIEN-36 | DCA shortCircuit parameter is not propagated by the keeper

Fixed 

Severity:

Low

Probability:

Unlikely

Impact:

Low

Path:

keeper-go/cmd/keeper/abi.go#L308

Description:

The keeper builds **DcaExecutionParams** with **ShortCircuit** hardcoded to **false**, regardless of the intended DCA configuration.

```
params := DcaExecutionParams{
    IntervalSeconds: interval,
    TotalExecutions: totalExec,
    TotalAmountIn:  totalAmt,
    FlexibilityBps:  uint16(order.FlexibilityBps),
    ShortCircuit:   false,
    Routes:         routes,
}
```

The keeper-facing pending order model also does not include a **shortCircuit** field, so the value cannot be passed through the current API response into calldata construction.

```
type PendingOrder struct {
    OrderHash      string json:"orderHash"
    OrderType      string json:"orderType"
    Signature      string json:"signature"
    IntervalSeconds string json:"intervalSeconds,omitempty"
    TotalExecutions string json:"totalExecutions,omitempty"
    TotalAmountIn  string json:"totalAmountIn,omitempty"
    FlexibilityBps  int    json:"flexibilityBps,omitempty"
    TrailPercent   int    json:"trailPercent,omitempty"
    DesignatedKeeper string json:"designatedKeeper,omitempty"
}
```

On-chain, this parameter is stored during lazy initialization and controls whether the trigger condition is considered reached after the first successful trigger-based execution.

```

if (data.slot2 == 0 && !data.exists) {
    ...
    data.shortCircuit = params.shortCircuit;
}

...

if (order.triggerPrice > 0 && !data.triggerReached) {
    if (!_isOrderReady(order, currentPrice)) revert OrderNotReady();
    if (data.shortCircuit) data.triggerReached = true;
}

```

As a result, DCA orders initialized through the keeper path cannot enable the intended one-time trigger behavior. After the first execution, later executions will continue to re-check the trigger condition instead of using the stored **triggerReached** state.

Remediation:

- Add shortCircuit to the DCA order data model used by the indexer and keeper.
- Persist and return the field in the keeper pending-orders API.
- Use the order's stored shortCircuit value when building DcaExecutionParams.

ALIEN-28 | ExecuteTrusted lacks guard against Stop-Loss/DCA fills

Fixed 

Severity:

Low

Probability:

Unlikely

Impact:

Low

Description:

executeTrusted accepts any order with **trustKeeper=true** regardless of its type, skipping the invariants each specialized order type relies on. DCA orders have their interval schedule bypassed, a keeper can drain a slice immediately via **executeTrusted** while all scheduled DCA slices remain executable, resulting in **n+1** slices drained instead of **n**. Trailing stop orders can be executed before the HWM-based trigger is ever reached. Stop-loss and take-profit orders can be filled at any market price without **_isOrderReady** being checked.

Remediation:

Consider only executing standard orders:

```
if (order.makerTraits.allowMultipleFills()) revert NotAllowed();
if (order.makerTraits.isTrailing()) revert NotAllowed();
if (order.makerTraits.isStopLoss() || order.triggerPrice != 0) revert NotAllowed();
```

ALIEN-25 | Swap Surplus Logic Is Bypassable and Economically Flawed

Acknowledged

Severity:

Low

Probability:

Rare

Impact:

Low

Path:

`./contracts/EpsilonRouter.sol`

Description:

The **swap** function applies a surplus fee when **amountOut > expectedAmountOut**. Unlike the order-book flow where surplus represents genuine price improvement found by a keeper, in **swap** the user constructs their own routes and knows the exact output before submitting.

Additionally, a user can bypass the surplus capture entirely by passing **expectedAmountOut = 0** (excluded by the **> 0** guard) or **expectedAmountOut = type(uint256).max**.

Remediation:

Remove the surplus functionality in Swap.

ALIEN-24 | Post-hook receives gross DEX output, not net user amount

Fixed ✓

Severity:

Low

Probability:

Rare

Impact:

Low

Path:

./contracts/EpsilonRouter.sol

Description:

`_finalizeFill` calls `_tryPostHook` with the `amountOut` returned by `_executeSwapRoutes`, the gross output before any deductions:

```
function _finalizeFill(..., uint256 amountOut, ...) internal {
    (uint256 adjustedAmountOut, uint256 surplusCaptured) = _captureSurplus(..., amountOut, ...);
    _distributeFees(..., adjustedAmountOut);
    _tryPostHook(..., amountOut, surplusCaptured); // ← gross, not what user received
}
```

What the user actually receives is `amountOut - surplusProtocolShare - protocolFee - keeperFee - referralFee`. The hook has no way to derive this without querying `protocolFeeBps`, `_currentKeeperFeeBps`, `referralFeeBps`, `surplusSplitBps`, and `surplusCaptured`, since the `surplusCaptured` param only carries the protocol's surplus share, not the full picture.

Remediation:

Pass the net `userAmount` (the amount actually transferred to the receiver inside `_distributeFees`) to `_tryPostHook` instead of the gross `amountOut`.

ALIEN-7 | minExecutionValue Does Not scale with decimals

Fixed 

Severity:

Low

Probability:

Rare

Impact:

Low

Path:

contracts/EpsilonRouter.sol

Description:

The **EpsilonRouter** attempts to prevent spam via a fixed minimum order size:

```
uint256 public minExecutionValue = 100e6;  
  
function _validateOrderParams(Order calldata order) private view {  
    if (order.amountIn < minExecutionValue) revert ExecutionTooSmall();  
}
```

However, this value is not normalized across tokens.

For 6-decimal tokens (e.g., USDC/USDT), **100e6** equals to \$100, which may be unnecessarily restrictive for users placing smaller orders. At the same time, for higher-value tokens such as ETH, this minimum becomes relatively insignificant.

As a result, spam protection is inconsistently enforced depending on token decimals and market price, potentially restricting legitimate users.

Remediation:

Normalize **minExecutionValue** based on token decimals to ensure consistent minimum order sizes across tokens with different decimal configurations, instead of relying on a fixed value.

ALIEN-20 | Matcher fills can desynchronize indexed order status

Fixed 

Severity:

Low

Probability:

Rare

Impact:

Low

Path:

contracts/EpsilonRouter.sol#L637
contracts/EpsilonRouter.sol#L665
contracts/EpsilonMatcher.sol#L152

Description:

Matcher execution paths update on-chain invalidation state, but the emitted events do not consistently describe the resulting order remaining state. This can cause the indexer to keep already-matched orders as active, or mark partially fillable hybrid orders as filled.

In the pair and batch matcher paths, `matcherPullLeg()` invalidates the filled amount but does not emit `OrderFilled`.

```
uint256 remaining = _checkRemainingMakingAmount(order, orderHash);
if (fillAmount > remaining) revert MatchAmountTooLarge();

oraclePrice = OracleLib.getPriceWithFallback(priceVerifier, fallbackOracles, order.tokenIn,
order.tokenOut);
if (oraclePrice == 0) revert OrderNotReady();
if (order.triggerPrice != 0 && !_isOrderReady(order, oraclePrice)) revert OrderNotReady();

_invalidateOrder(order, orderHash, remaining, fillAmount, order.maker);
netToRecipient = _matcherDistribute(order.tokenIn, order.maker, recipient, fillAmount,
order.referrer, order.referralFeeBps, keeperRecipient);
```

`EpsilonMatcher` emits `OrdersMatched`, but the indexer only records the match and does not update either order's status or remaining amount.

```
(, uint256 oraclePrice) = _callMatcherPullLeg(sellOrder, sellSig, matchAmountA, buyRecv,
msg.sender, keeperFeeBps);

uint256 sellDec = 10 ** IERC20Metadata(sellOrder.tokenIn).decimals();
amountB = (matchAmountA * oraclePrice) / sellDec;
if (amountB == 0) revert InvalidAmount();
```

```

_callMatcherPullLeg(buyOrder, buySig, amountB, sellRecv, msg.sender, keeperFeeBps);

emit OrdersMatched(sellHash, buyHash, msg.sender, matchAmountA, amountB, oraclePrice);

```

As a result, orders that were matched and invalidated on-chain can remain pending or active off-chain. These stale rows may continue to appear in keeper and orderbook queries.

```

func (idx *Indexer) handleOrdersMatched(ctx context.Context, l types.Log) error {
    eventsProcessed.WithLabelValues("orders_matched").Inc()
    if len(l.Topics) < 4 || len(l.Data) < 96 {
        return nil
    }
    ...
    if _, err := idx.db.ExecContext(cctx,
        "INSERT INTO
order_matches(sell_order_hash,buy_order_hash,keeper,amount_a,amount_b,clearing_price,tx_has
h,block_number) VALUES(?,?,?,?,?,?,?,?,?)",
        sellHash, buyHash, keeper, amountA, amountB, clearingPrice, l.TxHash.Hex(),
l.BlockNumber); err != nil {
        return fmt.Errorf("insert match: %w", err)
    }
}

```

In the hybrid matcher path, `matcherExecuteHybridLeg()` invalidates the combined `ringAmount + overflowAmount`, but emits `OrderFilled` only for the overflow portion and hardcodes `remainingAmount` to zero.

```

uint256 totalFill = ringAmount + overflowAmount;
uint256 remaining = _checkRemainingMakingAmount(order, orderHash);
if (totalFill > remaining) revert MatchAmountTooLarge();

oraclePrice = OracleLib.getPriceWithFallback(priceVerifier, fallbackOracles, order.tokenIn,
order.tokenOut);
if (oraclePrice == 0) revert OrderNotReady();
if (order.triggerPrice != 0 && !_isOrderReady(order, oraclePrice)) revert OrderNotReady();

_invalidateOrder(order, orderHash, remaining, totalFill, order.maker);

ringNet = _matcherDistribute(order.tokenIn, order.maker, recipient, ringAmount,
order.referrer, order.referralFeeBps, keeperRecipient);

if (overflowAmount > 0) {

```

```

overflowOut = _executeSwapRoutes(order, overflowRoutes, overflowAmount);
bool skipValidation = _shouldSkipValidation(order, orderHash);
OracleLib.validateSwapOutput(priceVerifier, order.tokenIn, order.tokenOut,
overflowAmount, overflowOut, _effectiveSlippage(order), order.makerTraits.isStopLoss(),
skipValidation);
_finalizeFill(order, orderHash, order.maker, overflowAmount, overflowOut, oraclePrice);
emit OrderFilled(orderHash, order.maker, keeperRecipient, overflowAmount, overflowOut,
0);
}

```

The indexer treats **OrderFilled.remainingAmount == 0** as terminal.

```

remaining := new(big.Int).SetBytes(l.Data[64:96]).String()

if _, err := idx.db.ExecContext(cctx,
    "INSERT INTO
fills(order_hash,maker,keeper,amount_in,amount_out,remaining,tx_hash,block_number)
VALUES(?,?,?,?,?,?,?,?)",
    hash, maker, keeper, amtIn, amtOut, remaining, l.TxHash.Hex(), l.BlockNumber); err !=
nil {
    return fmt.Errorf("insert fill: %w", err)
}

status := "active"
if remaining == "0" {
    status = "filled"
}

```

This can mark a partially fillable hybrid order as filled off-chain even when residual notional remains on-chain. The impact is limited to indexed order status and keeper/orderbook discovery, while on-chain invalidation remains authoritative.

Remediation:

- Emit matcher fill events with the actual filled amount and post-fill remaining amount.
- Keep event semantics consistent across pair, batch, ring, and hybrid matcher paths.
- Update **OrdersMatched** indexing to adjust order status or remaining state.
- Avoid deriving terminal status from inaccurate event values.

ALIEN-14 | Referral Fee Deducted but Not Distributed When referrer Is Zero Address

Fixed 

Severity:

Low

Probability:

Rare

Impact:

Low

Description:

During `_distributeFees`, it calculates and subtracts the referral fee from the `userAmount`.

However, if the referral fee cannot actually be sent (because `order.referrer` is not set), the transfer is skipped, but the fee is still deducted from `userAmount`.

```
function _distributeFees(Order calldata order, uint256 makingAmount, uint256 amountOut, uint256
adjustedAmountOut) internal {
    uint256 amount = order.makerTraits.takeFeesInTokenIn() ? makingAmount : adjustedAmountOut;

    uint256 protocolFee;
    uint256 keeperFee;
    uint256 referralFee;
    uint256 userAmount;
    uint16 kFeeBps = _currentKeeperFeeBps;

    unchecked {
        protocolFee = (amount * protocolFeeBps) / BPS_DENOMINATOR;
        keeperFee = (amount * kFeeBps) / BPS_DENOMINATOR;
        referralFee = (amount * order.referralFeeBps) / BPS_DENOMINATOR;
        userAmount = amount - protocolFee - keeperFee - referralFee;
    }

    address feeToken = order.makerTraits.takeFeesInTokenIn() ? order.tokenIn : order.tokenOut;
    address receiver = order.receiver != address(0) ? order.receiver : order.maker;

    if (protocolFee > 0) feeToken.safeTransfer(feeCollector, protocolFee);
    if (keeperFee > 0) feeToken.safeTransfer(msg.sender, keeperFee);
    if (referralFee > 0 && order.referrer != address(0)) feeToken.safeTransfer(order.referrer, referralFee);
    order.tokenOut.safeTransfer(receiver, order.makerTraits.takeFeesInTokenIn() ? adjustedAmountOut :
userAmount);
}
```

Remediation:

Add a validation in `_validateOrderParams`:

```
if (order.referralFeeBps > 0 && order.referrer == address(0)) revert ZeroAddress();
```

ALIEN-19 | Inconsistent Fee Rounding Can Cause False Reverts

Fixed 

Severity:

Low

Probability:

Rare

Impact:

Low

Path:

./contracts/EpsilonMatcher.sol

Description:

The router calculates fees in `_matcherDistribute` using three separate integer divisions:

```
function _matcherDistribute(
    address token,
    address from,
    address to,
    uint256 amount,
    address referrer,
    uint16 referralFeeBps,
    address keeperRecipient
) private returns (uint256 netAmount) {
    if (referralFeeBps > maxReferralFeeBps) revert ReferralFeeTooHigh();
    token.safeTransferFrom(from, address(this), amount);
    uint256 pFee; uint256 kFee; uint256 rFee;
    uint16 kFeeBps = _currentKeeperFeeBps;
    unchecked {
        pFee = (amount * matchingFeeBps) / BPS_DENOMINATOR;
        kFee = (amount * kFeeBps) / BPS_DENOMINATOR;
        rFee = (amount * referralFeeBps) / BPS_DENOMINATOR;
    }

    if (pFee > 0) token.safeTransfer(feeCollector, pFee);
    if (kFee > 0) token.safeTransfer(keeperRecipient, kFee);
    if (rFee > 0 && referrer != address(0)) token.safeTransfer(referrer, rFee);
    unchecked { netAmount = amount - pFee - kFee - rFee; }
    token.safeTransfer(to, netAmount);
}
```

Because each division rounds down, the total deducted fees could be slightly smaller than if they were calculated in a single combined division.

However, the ring fairness check later computes the net received amount using one combined division:

```
uint256 totalFeeBps = _matchingFeeBps + _keeperFeeBps + orders[next].referralFeeBps;
uint256 netReceived;
unchecked { netReceived = amounts[next] - (amounts[next] * totalFeeBps) / BPS_DENOMINATOR; }
```

Due to integer rounding:

$$\text{floor}(x \cdot A) + \text{floor}(x \cdot B) + \text{floor}(x \cdot C) \leq \text{floor}(x \cdot (A+B+C))$$

This mismatch can cause the fairness check to be too strict, leading to false reverts of valid settlements because of a 1-3 Wei difference.

Remediation:

Because `_callMatcherPullLeg` already returns the `netToRecipient` after fees, this can be used instead of recalculating it again.

```
-- (, uint256 oraclePrice) = _callMatcherPullLeg(sellOrder, sellSig, matchAmountA, buyRecv, msg.sender,
keeperFeeBps);
++ (uint256 netAmount, uint256 oraclePrice) = _callMatcherPullLeg(sellOrder, sellSig, matchAmountA,
buyRecv, msg.sender, keeperFeeBps);
```

ALIEN-17 | /api/keeper/pending Can be blocked by unsupported or phantom signed rows

Fixed 

Severity:

Low

Probability:

Unlikely

Impact:

Low

Path:

indexer-go/cmd/indexer/api.go#L479

Description:

The POST `/api/orders` endpoint accepts caller-supplied `orderType` and `orderHash`, while signature verification is performed only over the signed Order fields. The submitted hash is normalized and stored, but it is not recomputed from the signed order before insertion.

```
if req.OrderType == "" || req.OrderHash == "" {
    s.writeError(c, http.StatusBadRequest, "BAD_REQUEST", "orderType and orderHash are required")
    return
}

normalizedHash, ok := parseOrderHash(req.OrderHash)
if !ok {
    s.writeError(c, http.StatusBadRequest, "INVALID_HASH", "orderHash must be 0x + 64 hex chars")
    return
}
req.OrderHash = normalizedHash

if err := verifyOrderSignature(&req, s.domainSeparator, s.idx.client); err != nil {
```

```
if _, err := s.idx.db.ExecContext(ctx, INSERT INTO
orders(order_hash,order_type,salt,maker,receiver,token_in,token_out,amount_in,trigger_price,
maker_traits,referrer,referral_fee_bps,signature,status,source,interval_seconds,total_execut
ions,total_amount_in,flexibility_bps,trail_percent,designated_keeper)
VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?,pending,'api',?,?,?,?,?),
req.OrderHash, req.OrderType, req.Order.Salt, req.Order.Maker, req.Order.Receiver,
req.Order.TokenIn, req.Order.TokenOut, req.Order.AmountIn, req.Order.TriggerPrice,
req.Order.MakerTraits, req.Order.Referrer, req.Order.ReferralFeeBps, req.Signature,
req.IntervalSeconds, req.TotalExecutions, req.TotalAmountIn, req.FlexibilityBps,
req.TrailPercent, req.DesignatedKeeper); err != nil {
```


The `/api/keeper/pending` endpoint returns only the oldest 100 signed pending or active rows, without pagination or filtering by executable order type.

```
rows, err := s.idx.db.QueryContext(ctx, "SELECT
order_hash,order_type,COALESCE(salt,''),maker,COALESCE(receiver,''),COALESCE(token_in,''),CO
ALESCE(token_out,''),COALESCE(amount_in,''),COALESCE(trigger_price,''),COALESCE(maker_traits
'),COALESCE(referrer,''),COALESCE(referral_fee_bps,0),signature,COALESCE(interval_seconds,
'),COALESCE(total_executions,''),COALESCE(total_amount_in,''),COALESCE(flexibility_bps,0),C
OALESCE(trail_percent,0),COALESCE(designated_keeper,'') FROM orders WHERE status IN
('pending','active') AND signature!="" ORDER BY created_at LIMIT 100")
```

The keeper processes only the returned rows. Unknown order types are logged but are not marked as processed or removed from the pending set.

```
switch order.OrderType {
case "limit", "stop_loss", "take_profit":
    k.executeLimitOrder(ctx, order)
case "dca":
    k.executeDcaOrder(ctx, order)
case "trailing":
    k.executeTrailingOrder(ctx, order)
default:
    k.logger.Warn("unknown order type", "type", order.OrderType, "hash",
shortHash(order.OrderHash))
}
```

As a result, unsupported rows or rows stored under phantom hashes can occupy the first pending page. This does not make invalid orders executable on-chain, but it can delay later valid orders from being seen by the canonical keeper.

Remediation:

- Recompute and store orderHash from the submitted signed order.
- Reject unsupported orderType values during order submission.
- Add pagination or cursor-based fetching to the pending endpoint.
- Exclude non-executable rows from keeper pending results.
- Mark unsupported or permanently invalid rows with a terminal status.

ALIEN-18 | Mismatched setProtocolFee Parameters Between Manager and Router

Fixed 

Severity:

Informational

Probability:

Rare

Impact:

Informational

Path:

./contracts/EpsilonManager.sol

Description:

The **EpsilonRouter** defines **setProtocolFee** with 2 parameters:

```
function setProtocolFee(uint16 feeBps, address collector) external onlyOwnerOrManager
```

However, **EpsilonManager** calls it with 3 parameters:

```
function setProtocolFee(uint16 feeBps, uint16 keeperFeeBps, address collector) external onlyOwner {  
    router.setProtocolFee(feeBps, keeperFeeBps, collector);  
}
```

as defined in **IEpsilonRouterAdmin**:

```
interface IEpsilonRouterAdmin {  
    function setProtocolFee(uint16 feeBps, uint16 keeperFeeBps, address collector) external;  
}
```

The call will revert due to function signature mismatch.

Remediation:

Ensure the router implementation, interface, and manager all use the same parameters used in the router.

ALIEN-16 | Bit-Invalidator cancellations are not reflected in off-chain order status

Fixed 

Severity:

Informational

Probability:

Unlikely

Impact:

Informational

Path:

contracts/EpsilonRouter.sol#L711

Description:

`cancelOrder()` uses different events depending on the invalidation mode. Orders using the bit invalidator emit **BitInvalidatorUpdated**, while only non-bit invalidator cancellations emit **OrderCancelled**.

```
function cancelOrder(MakerTraits makerTraits, bytes32 orderHash) public {
    if (makerTraits.useBitInvalidator()) {
        uint256 inv;
        unchecked {
            inv = _bitInvalidator[msg.sender].massInvalidate(makerTraits.nonce(), 0);
        }
        emit BitInvalidatorUpdated(msg.sender, makerTraits.nonce() >> 8, inv);
    }
    else {
        _remainingInvalidator[msg.sender][orderHash] =
        RemainingInvalidatorLib.fullyFilled();
        emit OrderCancelled(orderHash);
    }
}
```

The indexer subscribes to and handles **OrderCancelled**, but does not subscribe to **BitInvalidatorUpdated**.

```
Topics: [][]common.Hash{{
    OrderRegisteredSig, OrderFilledSig, OrderCancelledSig, SwappedSig,
    SurplusCapturedSig, PostHookExecutedSig, PostHookSetSig,
    GlobalSurplusToggledSig, OrdersMatchedSig, RingMatchedSig,
}},
```

```

switch l.Topics[0] {
case OrderRegisteredSig:
    return idx.handleOrderRegistered(ctx, l)
case OrderFilledSig:
    return idx.handleOrderFilled(ctx, l)
case OrderCancelledSig:
    return idx.handleOrderCancelled(ctx, l)
}

```

`handleOrderCancelled()` updates the order status by `order_hash`, but this path is never reached for bit-invalidator cancellations.

```

func (idx Indexer) handleOrderCancelled(ctx context.Context, l types.Log) error {
    eventsProcessed.WithLabelValues("cancelled").Inc()
    if len(l.Topics) < 2 {
        return nil
    }
    cctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()
    hash := l.Topics[1].Hex()
    if _, err := idx.db.ExecContext(cctx, "UPDATE orders SET
status='cancelled',updated_at=CURRENT_TIMESTAMP WHERE order_hash=?", hash); err != nil {
        return fmt.Errorf("cancel: %w", err)
    }
}

```

As a result, signed orders cancelled through the bit invalidator can remain pending or active in the off-chain database even though they are no longer executable on-chain. The keeper-facing pending endpoint then continues to return those rows.

```

rows, err := s.idx.db.QueryContext(ctx, "SELECT
order_hash,order_type,COALESCE(salt,''),maker,COALESCE(receiver,''),COALESCE(token_in,''),CO
ALESCE(token_out,''),COALESCE(amount_in,''),COALESCE(trigger_price,''),COALESCE(maker_traits
',''),COALESCE(referrer,''),COALESCE(referral_fee_bps,0),signature,COALESCE(interval_seconds,
'),COALESCE(total_executions,''),COALESCE(total_amount_in,''),COALESCE(flexibility_bps,0),C
OALESCE(trail_percent,0),COALESCE(designated_keeper,'') FROM orders WHERE status IN
('pending','active') AND signature!="" ORDER BY created_at LIMIT 100")

```

This does not make the cancelled orders fillable on-chain, but it can cause stale cancelled rows to remain in keeper automation queues and delay processing of later valid orders.

Remediation:

- Subscribe to and process BitInvalidatorUpdated events in the indexer.
- Reconcile affected pending orders by maker and invalidator slot.
- Exclude bit-invalidated orders from keeper pending queries.

ALIEN-11 | Redundant Order Creation in executeTrailingStop()

Fixed 

Severity:

Informational

Probability:

Rare

Impact:

Informational

Path:

contracts/EpsilonRouter.sol#L416-L429

Description:

In `executeTrailingStop()`, lines 416–429 allow the caller to create the order for the maker if it does not already exist:

```
if (!data.exists) {
    _verifySignature(maker, orderHash, signature);
    if (!order.makerTraits.isTrailing()) revert NotTrailingOrder();
    if (params.trailPercent == 0 || params.trailPercent > 10000) revert InvalidTrailPercent();
    if (params.designatedKeeper == address(0)) revert ZeroAddress();

    unchecked {
        data.slot1 = uint256(params.trailPercent) << TRAIL_PERCENT_OFFSET;
    }
    data.exists = true;
    _orderCreator[orderHash] = maker;
    designatedKeeper[orderHash] = params.designatedKeeper;
    keeper = params.designatedKeeper;
}
```

However, this creation logic is redundant due to the initialized bit check at lines 435–437:

```
unchecked {
    if ((slot1 & (uint256(1) << TRAIL_INITIALIZED_BIT /*144*/)) == 0) revert NotInitialized();
}
```

This bit can only be set via `updateTrailingStopHighWaterMark()`. Therefore, any order created within `executeTrailingStop()` will not have this bit set.

As a result, the function will always revert when attempting to execute a newly created order within `executeTrailingStop()`.

Remediation:

Consider removing the order creation logic from `executeTrailingStop()`.

ALIEN-1 | Redundant oraclePrice == 0 Check

Fixed ✓

Severity:

Informational

Probability:

Rare

Impact:

Informational

Path:

/contracts/EpsilonRouter.sol

Description:

```
oraclePrice = OracleLib.getPriceWithFallback(priceVerifier, fallbackOracles, order.tokenIn, order.tokenOut);  
if (oraclePrice == 0) revert OrderNotReady();
```

The check for `oraclePrice == 0` is always false.

`getPriceWithFallback()` never returns 0:

- If a valid price exists → it returns a non-zero value
- If no price is available → it reverts (`NoOracleAvailable`)

```
function getPriceWithFallback(...) internal returns (uint256 price) {  
  -- Snip  
  price = _tryFallbackAdapters(fallbackAdapters, tokenIn, tokenOut);  
  if (price != 0) return price;  
  
  revert NoOracleAvailable();  
}
```

Remediation:

Remove the check entirely.

ALIEN-26 | Reuse remaining instead of recalculating completion condition

Fixed 

Severity:

Informational

Probability:

Rare

Impact:

Informational

Path:

./contracts/EpsilonRouter.sol

Description:

In `executeDcaOrder`, the expression `executionsCompleted + 1` is computed multiple times within the same `unchecked` block:

```
function executeDcaOrder(Order calldata order, bytes calldata signature, DcaExecutionParams calldata params, uint16 keeperFeeBps) external nonReentrant whenNotPaused returns (uint256 amountOut) {  
    -- SNIP --  
    unchecked {  
        uint256 remaining = storedTotalExecutions - (executionsCompleted + 1);  
        emit OrderFilled(orderHash, maker, msg.sender, makingAmount, amountOut, remaining);  
        if (executionsCompleted + 1 >= storedTotalExecutions) { delete orderData[orderHash]; delete  
_orderCreator[orderHash]; }  
    }  
}
```

Since `executionsCompleted + 1` is already implicitly used to derive `remaining`, recomputing it again in the conditional introduces unnecessary arithmetic and reduces readability.

Remediation:

This can be simplified by reusing the computed `remaining` value:

```
uint256 remaining = storedTotalExecutions - (executionsCompleted + 1);  
emit OrderFilled(orderHash, maker, msg.sender, makingAmount, amountOut, remaining);  
-- if (executionsCompleted + 1 >= storedTotalExecutions) {  
++ if (remaining == 0) {  
++   delete orderData[orderHash];  
++   delete _orderCreator[orderHash];  
++}
```

ALIEN-15 | Stop-Loss orders with maxslippageBps Above 5000 Will Revert with Underflow

Fixed 

Severity:

Informational

Probability:

Rare

Impact:

Informational

Path:

./contracts/EpsilonRouter.sol

Description:

Stop-loss orders apply a 2× multiplier to the maker's slippage tolerance to account for volatile market conditions, since the **BPS_DENOMINATOR** is 10_000, any slippage above 5000 will revert with underflow.

```
function validateSwapOutput() external view {
  -- snip
  uint256 maxSlippage = maxSlippageOverride > 0 ? maxSlippageOverride : maxSlippageBps;
  if (isStopLoss) maxSlippage = maxSlippage * 2;

  if (useTWAP) {
    // Use TWAP oracle for validation
    twapOracle.validateSwapOutput(tokenIn, tokenOut, amountIn, amountOut, maxSlippage);
  } else {
    // Use Redstone for validation
    _validateWithRedstone(tokenIn, tokenOut, amountIn, amountOut, maxSlippage);
  }
}
```

→ Underflows in EpsilonTWAPOracle

```
function validateSwapOutput(
  address tokenIn,
  address tokenOut,
  uint256 amountIn,
  uint256 amountOut,
  uint256 maxSlippageBps
) external view {
  -- SNIP
  uint256 minAcceptable = FullMath.mulDiv(expectedOut, BPS_DENOMINATOR - maxSlippageBps,
    BPS_DENOMINATOR);
```

Remediation:

Validate the field in `TraitsLib.validate()`:

```
if (self.maxSlippageBps() > 5000 && self.isStopLoss()) revert InvalidTraitsCombination();
```

hexens x  ALIEN BASE

