Security Audit Report

Levery Ethereum

Delivered: October 3rd, 2025

Prepared for Levery Inc. by

runtime verification

Table of Contents

Disclaimer

Executive Summary

Levery Inc. engaged Runtime Verification Inc. to conduct a security audit of the Levery design and the associated smart contract code. The objective was to review the business logic and implementation in Solidity and identify any issues that could cause the system to malfunction or be vulnerable to exploitation.

Levery is a regulatory-grade AMM protocol built on Uniswap V4 that bridges regulated capital with non-custodial liquidity. The protocol integrates institutional-grade permissioning with DeFi efficiency through granular KYC/AML controls via an on-chain PermissionManager for role-based access. It features dynamic, oracle-driven fees that adapt to market deviations, protecting liquidity providers from toxic arbitrage flows. It utilizes soulbound liquidity positions through non-transferable ERC-721 tokens for robust compliance governance. The system includes a service fee vault with transparent revenue-sharing mechanisms and emergency pause controls at global and per-pool levels. Levery is designed to meet institutional requirements for on-chain participation while maintaining DeFi's capital efficiency, targeting the gap between traditional finance's compliance needs and decentralized finance's accessibility.

The audit was conducted over four and a half calendar weeks, from July 16th to August 15th. Runtime Verification performed a design review to assess the protocol's high-level intent and security-critical invariants, followed by a focused manual review of the Solidity implementation. We used Kontrol, our formal verification tool, to support this process by specifying and checking invariants across symbolic state transitions.

Scope

This audit covers only the code contained in a client-provided, currently private GitHub repository. Within this repository, specific files and contracts were highlighted as being in scope for this engagement. The repositories, contracts, and commit information are detailed below:

Levery GitHub Repository (private as of writing this)
https://github.com/levery-org/levery-contracts
Commit: 970ec76

The following files were in the scope of the audit:

- src/Levery.sol
- src/CompliantRouter.sol
- src/SoulboundPositionManager.sol
- src/utils/PermissionManager.sol
- src/utils/BaseSwapRouterPermit2.sol
- src/utils/PositionDescriptor.sol
- src/libraries/Descriptor.sol

The audit is limited to the artifacts listed above. Off-chain components, third-party dependencies, deployment and upgrade scripts, and any client-side logic are excluded from the scope of this engagement.

Our security analysis is based on the following operational assumptions. If any of these assumptions are violated, the protocol’s security guarantees may no longer hold and additional review may be required.

- Base Protocol Correctness
  We assume that the underlying protocols, such as Uniswap and Chainlink, behave as specified and do not contain critical vulnerabilities.
- Trusted Admin Addresses
  All addresses holding admin roles are trusted, properly secured, and act in good faith.
- Price Oracle Integrity
  Price feeds (oracles) report accurate asset valuations and cannot be tampered with to manipulate the deviation between the pool prices of the tokens and the market price.

- Collateral Curation & Systemic-Risk Mitigation
  Adequate processes for monitoring, setting up market configuration data, updating market configuration parameters, and classifying collateral (e.g., implementing tiered collateral system, liquidity thresholds, ongoing asset vetting, LTVs) are in place to prevent systemic failures.

Methodology

Although manual code review cannot guarantee the discovery of all possible security vulnerabilities, as noted in our Disclaimer, we followed a systematic approach to make this audit as thorough and impactful as possible within the allotted timeframe.

The audit engagement lasted four-and-a-half weeks, from July 16th to August 15th, and began with a focused design review. We allocated the first week to analyze the architecture and intended functionality of the Levery system. This included reasoning about the interactions between the protocol components and identifying properties that should be upheld throughout the system's lifecycle. Following the design review, we conducted a thorough manual code review of the in-scope contracts, progressing systematically.

The engagement included specialized focus areas such as interest rate strategies with underlying protocol integrations (Uniswap) and price oracle analysis covering integrations with multiple providers (Chainlink, Scribe, Redstone).

This process was aided by our formal verification tool, Kontrol, which enables symbolic execution of Solidity code. Where appropriate, we defined formal properties and used Kontrol to verify them under various symbolic inputs and system states.

Findings presented in this report stem from a combination of:

- Manual inspection of the Solidity source code.
- Design-level reasoning about contract interactions and system invariants.
- Symbolic proofs and property-based assertions were executed using Kontrol.

In addition to identifying bugs and vulnerabilities, we also evaluated gas usage patterns, reviewed edge-case handling, and provided recommendations for code clarity and safety improvements.

Throughout the engagement, we held internal discussions among auditors to cross-review the findings and validate risk assessments.

Design Assumptions and Architectural Decisions

This section describes design decisions that reflect the institutional deployment context and establish the foundation for maintaining security throughout the system.

Institutional Deployment

Levery operates under a controlled deployment paradigm, provisioning critical infrastructure components through internal operations with established security procedures. The constructor validation approach implements non-zero address checks without `addr.code.length` validation, following Uniswap v4 periphery patterns and assuming trusted deployer infrastructure where addresses are controlled through operational procedures.

Oracle Management

Levery's original design delegated oracle operations to externally managed wallets without protocol-level staleness validation. The `getLastOraclePrice` function prioritized operational flexibility across multiple providers (Chainlink, ChronicleLabs's Scribe, Redstone, and Pyth) through the standard `AggregatorV3Interface`, but it lacked freshness controls.

During review, issue A01: Levery.getLastOraclePrice does not check if the price is stale. identified that stale oracle prices could cause usability degradation, such as swaps being rejected due to incorrectly calculated dynamic fees.

The enhanced design ensures that pools connect to reputable price oracles and, by default, enforce freshness via heartbeat and `maxAge` bounds. For event-driven or domain-specific feeds (e.g., certain RWAs), Levery allows pool-level configuration without heartbeat requirements; in such cases, the institution documents the update policy and accepts that dynamic LP-fee adjustments may be constrained when the price is considered stale by policy.

Compliance Integration

The fixed 32-byte hookData format uses exactly the user address for KYC verification, with routers tolerating ≥32 bytes for forward compatibility, but hooks decoding only the address portion. This provides a simple, deterministic compliance path with minimal calldata overhead.

Process-based institutional branding validation reflects the assumption that administrative content sources are trusted and validated through separate institutional procedures rather than requiring comprehensive on-chain sanitization.

Fee Architecture

Levery distinguishes the LP Fee (liquidity providers' fee) from the Service Fee (institution fee). The service fee is denominated in the input token. For exact-input swaps, it is collected in `beforeSwap`; for exact-output swaps, it is collected in `afterSwap`, still in the input currency. LP fee (including any dynamic adjustment) remains orthogonal to the service fee.

System Flow

Users interact through either CompliantRouter (for swaps) or SoulboundPositionManager (for liquidity operations), executing through PoolManager, which triggers Levery hooks. All swaps and liquidity actions pass through Levery's hooks for compliance validation, role checking, dynamic fee calculation via oracle integration, and service fee collection to the PaymentSplitter vault.

Formal Model and Invariant Properties

The protocol's security relies on several mathematically precise invariants that must hold across all system states. These formal properties provide the foundation for our security analysis and guided our verification efforts using symbolic execution.

Formal Definitions

Valid Swap

A swap is valid if and only if:

- `sender ∈ {authorized swapRouter, quoter}`.
- `permissionManager.isSwapAllowed(user) = true`.
- If a pool-required role exists, the user holds that role with swap permission.
- The pool and contract are not paused.
- If an oracle is set for the pool, the oracle price is nonzero.

Valid Liquidity Operation

A liquidity operation is valid if and only if:

- `sender == authorized positionManager`.
- `permissionManager.isLiquidityAllowed(owner) = true`.
- If the pool requires a role, the owner holds that role with liquidity permission.
- The pool and contract are not paused.

Valid Position Modification

A position modification is valid if and only if:

- `caller == ownerOf(tokenId)`.
- `sender == authorized positionManager`.
- `permissionManager.isLiquidityAllowed(owner) = true`.
- The pool and contract are not paused.
- All slippage and compliance checks pass.

Dynamic LP Fee

For each swap in pool p, the dynamic fee is calculated as:

$$\mathrm{dynamicFee}(p) = \mathrm{baseFee}(p) + \left(\frac{|P_{\mathrm{onchain}} - P_{\mathrm{oracle}}|}{P_{\mathrm{oracle}}} \times \mathrm{deviationFeeFactor} \right)$$

where all fees are bounded by `[0, MAX_PPM]`.

Role Assignment
A user u has a role r for pool p if $\mathrm{rolePermissions}(r, u)$ includes the required action and $\mathrm{poolRequiredRole}(p) = r$.

Service Fee Deduction

- For exact-input swaps, deduct from the input token amount and send to the Fee Vault before swap.
- For exact-output swaps, deduct from the input amount after the swap and send to the Fee Vault

Soulbound Position
A position token t is soulbound if all transfer and approval functions revert, `ownerOf(t)` is set at mint and never changes, and only `ownerOf(t)` can modify or burn t.

Emergency Pause State
A pool or the contract is paused if its corresponding paused flag is set. While paused, no user actions are valid, except for explicitly allowed admin/emergency actions.

Admin Authority
Levery implements a dual-authority model distinguishing protocol-level and venue-level administration. The provider holds protocol-level authority, including global pause/unpause capabilities, critical address updates for core infrastructure components (router, quoter, positionManager, permissionManager, fee vault), and global serviceFee configuration.

The institution operates at the venue level with authority over fee policy (global and per-pool baseFee), oracle assignments, individual pool pause controls, role definitions, and user permission assignment per pool. The PermissionManager admin role is held by the institution or an address it designates, enabling decentralized permission management while maintaining clear operational boundaries between protocol infrastructure and venue-specific policies.

Core Invariants

Compliance Enforcement

- No user can swap or manage liquidity unless permitted by the PermissionManager and, if required, by role.
- Only the current admin can update permissions or change the admin.
- Pool creation is restricted to institution-level privileges (enforced in `beforeInitialize`).
- Oracle management requires institution-level access with valid address validation.

Fee and Oracle

- All fees are within `[0, MAX_PPM]` with mathematical validation.
- The dynamic fee is monotonic in price deviation: higher deviations result in higher fees.
- Swaps revert if the oracle price is zero.
- Service fees are always deducted and sent to the Fee Vault when swaps succeed.
- When paused, swaps and liquidity actions must revert.

Soulbound Position

- All ERC-721 transfer and approval entry points for position tokens revert: `transferFrom`, both `safeTransferFrom` overloads, `approve`, and `setApprovalForAll`.
- Liquidity can only be managed via the authorized `positionManager`.
- Ownership of a position token never changes after mint.
- Only `ownerOf(tokenId)` can modify or burn a position; no operator approvals are permitted.
- Each `tokenId` is unique and never reused.

Role Management

- Roles cannot be created twice or assigned/revoked if they do not exist.
- Pool-required roles must reference an existing role.
- Role permissions are properly enforced for pool access.

Operational Properties

Safety Properties ensure that bad things never happen:

- Compliance is always enforced: no swap or liquidity operation can be executed unless the user passes compliance and role checks.
- Fee bounds are maintained: all fees remain within `[0, MAX_PPM]`.

- Oracle price validation: if oracle price is zero, swap reverts.
- Pausing effectiveness: when paused, all user operations revert except allowed admin/emergency actions.
- Pool integrity: pool balances are always non-negative, and total supply matches the sum of individual holdings.
- Non-transferability: soulbound positions cannot be transferred under any circumstances.
- Self-custody guarantee: neither provider nor institution takes custody of user assets. Current regulatory compliance is enforced through pool-level controls and permission management. For future regulatory requirements, individual positions may be frozen for compliance purposes, but can never be transferred or redeemed by third parties; only the owner wallet can unwind once conditions are met.

Liveness Properties ensure that good things eventually happen:

- It should always be possible for a compliant, authorized user to perform a swap or liquidity action, provided the contract and pool are not paused and all invariants are satisfied.
- If a user is granted permission and the appropriate role, they will eventually be able to execute the corresponding action, assuming the contract and pool remain unpaused.
- It should always be possible for the owner of a soulbound position to close and withdraw liquidity, provided the contract and pool are not paused and all checks pass.
- If the institution initiates a valid update (creating a pool, assigning a role, setting an oracle), that update will eventually take effect, provided the contract is not paused and all invariants are respected.
- If a paused state is lifted, all compliant, authorized users regain the ability to interact with the protocol.
- If a payee in the payment splitter has a positive releasable balance, they can always claim their funds, provided the contract holds a sufficient balance and no admin action removes their shares.

Symbolic Execution with Kontrol

In addition to the code review, we have adapted some of the tests in the repository for symbolic execution with Runtime Verification's formal verification tool, Kontrol. Kontrol is designed to integrate seamlessly with Solidity-based projects. It enables developers to write property-based tests in Solidity and leverage symbolic execution to verify them, thus ensuring that smart contracts behave as intended under all possible inputs and scenarios.

Unlike traditional testing approaches that use concrete values or fuzzing with random inputs, Kontrol interprets test parameters as symbolic variables and employs mathematical reasoning to explore all execution paths simultaneously. This comprehensive coverage ensures properties hold across the entire input space rather than just specific test cases. For proofs that are failing, Kontrol produces the model, or the counterexample, with concrete assignments to symbolic variables that trigger the execution path which causes the failure.

Note that in practice, some assumptions need to be made: to eliminate impossible initial states, to exclude properties that do not apply, or to avoid corner cases that complicate symbolic execution. A typical example is to bound the value of test inputs and storage variables to avoid reverts due to overflows.

Preparing Kontrol proofs

To support reasoning about conversions between signed integers and fixed-width 32-byte words, we extended Kontrol with new simplification rules for EVM arithmetic. These improvements allow the engine to efficiently handle operations like sign extension (used for type casting) and modular arithmetic (which corresponds to `powmod`, `chop`, and `signextend` in K), which are central to Uniswap V4’s dynamic fee computation and core mathematical routines.

Additionally, to facilitate the reasoning, in the `setUp` function used by Kontrol tests, we deploy the `Levery` hook at the hardcoded address, which satisfies the address requirements in accordance with the hook functions it implements.

We created a separate `test/kontrol` directory with modified contract copies to preserve the original sources. This directory includes selected tests from `LeveryFuzz` and `AccessControl` suites, with access control tests generalized through input parameterization for symbolic execution and fuzzing.

```diff
diff --git a/test/flows/AccessControl.t.sol b/test/flows/AccessControl.t.sol
--- a/test/flows/AccessControl.t.sol
+++ b/test/kontrol/LeveryProve.k.sol

-    function test_updateBaseFee_byInstitution() public {
-        uint24 newBaseFee = 4321;
+    function testFuzz_updateBaseFee_byInstitution(uint24 newBaseFee) public {
+        vm.assume(newBaseFee <= MAX_PPM);
         levery.updateBaseFee(newBaseFee);
         assertEq(levery.baseFee(), newBaseFee);
     }

-    function test_setPoolBaseFee_byInstitution() public {
-        uint24 fee = 250;
-        levery.setPoolBaseFee(key, fee);
+    function testFuzz_setPoolBaseFee_byInstitution(uint24 newFee) public {
+        vm.assume(newFee <= MAX_PPM);
+        levery.setPoolBaseFee(key, newFee);
         bytes32 pid = PoolId.unwrap(key.toId());
-        assertEq(levery.poolBaseFees(pid), fee);
+        assertEq(levery.poolBaseFees(pid), newFee);
     }
```

These tests focus on configuration management and access control, validating core system properties including fee parameter bounds, input validation, and oracle configuration integrity. Specifically, the tests verify that all fee parameters are constrained within the valid range of zero to one million parts per million (defined as `MAX_PPM`), that invalid inputs trigger appropriate revert conditions, and that oracle configurations can be established and retrieved for pools with any non-zero oracle address and the associated comparison flag.

The results of the tests are available in Kaas Report.

Reproducing the proofs

To reproduce the results of this verification locally, follow the steps below:

1. Install Kontrol

```
bash <(curl https://kframework.org/install)
kup install kontrol
```

2. Run the proofs

```
export FOUNDRY_PROFILE=kontrol-proofs
kontrol build
kontrol prove
```

The `kontrol-proofs` Foundry profile is configured in the `foundry.toml` file to build the project with Kontrol. The `kontrol.toml` file contains a set of options and flags that Kontrol will use during execution. Users can edit the file to change these options or turn flags on or off. Refer to the Kontrol documentation to learn more about Kontrol options.

Findings

This section contains all issues identified during the audit that could lead to unintended behavior, security vulnerabilities, or failure to enforce the protocol’s intended logic. Each issue is documented with a description, potential impact, and recommended remediation steps.

A01: Levery.getLastOraclePrice does not check if the price is stale.

Severity: Medium Difficulty: Medium Recommended Action: Fix Code

Partially addressed by client

Oracle data feeds can return stale pricing data for a variety of reasons. If oracle data is stale, dynamic fee calculation may degrade execution quality, using incorrect fees or cause swaps to revert. Smart contracts should always check the `updatedAt` parameter returned by `latestRoundData()` and compare it to a staleness threshold. The staleness threshold should correspond to the heartbeat of the oracle's price feed. This can be found on Chainlink's list of Ethereum mainnet price feeds by checking the "Show More Details" box, which will show the "Heartbeat" column for each feed. (source)

runtimeverification/_audits_levery-org_levery-contracts/src/Levery.sol

Line 319 in 970ec76

```
319          (, int256 answer,,,) = priceFeed.latestRoundData();
```

Recommendation:

Implement heartbeat checks.

Consider implementing range validation to catch obviously bad prices that pass staleness checks. That way, it would work across different oracle sources.

Implement provider-specific validations that accommodate different heartbeat patterns and staleness thresholds for each of the supported oracle providers (Chainlink, Scribe, Redstone, Pyth).

Analysis

The Levery contract assigns an oracle for each pool using the hooks. Multiple providers, such as Chainlink, Chronicle Labs' Scribe, Redstone, and Pyth, are supported, accessed through a common interface where applicable (e.g., `AggregatorV3Interface`).

```
    struct PoolOracle {
        address oracle;
        bool compareWithPrice0;
    }
    mapping(bytes32 => PoolOracle) public poolOracles;
```

However, due to this generality, determining the staleness duration (e.g., Chainlink's heartbeat) is not straightforward.

Client Response

A stale oracle can bias dynamic LP-fee adjustments, but it does not compromise pool accounting or custody. In the worst case, LP fees are misestimated; base fees remain in effect.

For A01, we would like to clarify that in our specific architecture, a stale oracle price does not pose a “potential loss of funds for the user and/or the protocol” as mentioned in the draft report. The worst-case impact is limited to usability degradation, for example, swaps being rejected due to higher calculated dynamic fees, or the system falling back to the default Uniswap V4 base fee behavior.

That said, we still intend to improve the implementation by adding an optional heartbeat check per oracle feed. When configured, the system will ignore dynamic fee adjustments from outdated prices, while maintaining compatibility with event-driven or custom oracle sources.

Status

In commit 4017c15c6341c461cf6c4fb7d0ce546f3968aba8, there is a new `maxAge` field added to the `PoolOracle` structure.

A new `(maxAge > 0 && block.timestamp > updatedAt + maxAge)` check has been added to ensure the price is not stale. This check can be bypassed by assigning 0 to `maxAge`.

A02: Oracle negative price cast in '_adjustSwapFee'

Severity: Low | Difficulty: High | Recommended Action: Fix Code | Addressed by client

The value of the price returned by the `latestRoundData` is of the `int256` type and, in _adjustSwapFee, it is directly cast to the `uint256` type without first checking if it's a negative value. It is followed by a zero check after the cast is done. However, if an oracle returned `-1`, for example, the `uint256` cast would change it to `2 ** 256 - 1`, and the zero check would be redundant.

runtimeverification/_audits_levery-org_levery-contracts/src/Levery.sol

Line 393 to 394 in 970ec76

```
        uint256 mp = uint256(getLastOraclePrice(po.oracle, key));
        if (mp == 0) revert OraclePriceZero();
```

The `int256` type choice for the answer field in the `AggregatorV3Interface` is intentional to support various types of data feeds beyond simple asset prices. Most production price feeds for assets like `ETH/USD` or `LINK/USD` should never return negative values, but the interface design allows flexibility for different oracle use cases.

Recommendation

Change the zero check to a negative number check and consider moving this check and the `uint256` casting inside `getLastOraclePrice`.

```
    function getLastOraclePrice(address _oracle, PoolKey calldata key) public view override
returns (uint256) {
        AggregatorV3Interface priceFeed = AggregatorV3Interface(_oracle);
        uint8 feedDecimals = priceFeed.decimals();
        (, int256 answer,,,) = priceFeed.latestRoundData();
        if(answer <= 0) revert OraclePriceZero(); // Add this here
        ...
        return uint256(answer); //uint256 cast here
    }
```

Status

Addressed in commit 81198fb2e9b0527b713f3240b67f377d3f6a02d5.

A03: Missing validation of permitted.token against expected input currency

Severity: Low | Difficulty: Medium | Recommended Action: Fix Code | Addressed by client

The router fails to verify that the token specified in the Permit2 payload matches the input currency expected by the swap. An attacker could supply a permit for an arbitrary token, causing the router to pull a different token from the user than the one used for the swap, potentially executing a swap without supplying the required input token.

Recommendation

Validate that `permitPayload.permit.permitted.token` matches the address of `inputCurrency` (or its ERC-20 address) before invoking `permitTransferFrom` or `transferFrom`, and revert the transaction if they differ.

Notes

This finding was identified with the assistance of the Almanax tool.

Auditor notes

This is a low-severity denial-of-service (DoS) vulnerability, not a critical loss-of-funds issue. The locking mechanism protects against token confusion attacks by ensuring that exact debts are paid in the correct currencies after the `unlockCallback` function finishes executing, so incorrect token transfers don't satisfy the debt.

```
    /// @inheritdoc IPoolManager
    function unlock(bytes calldata data) external override returns (bytes memory result) {
        if (Lock.isUnlocked()) AlreadyUnlocked.selector.revertWith();

        Lock.unlock();

        // the caller does everything in this callback, including paying what they owe via
calls to settle
        result = IUnlockCallback(msg.sender).unlockCallback(data);
        if (NonzeroDeltaCount.read() != 0) CurrencyNotSettled.selector.revertWith();
        Lock.lock();
    }
```

The attack flow:

1. Swap executes with expected `inputCurrency`, creating a debt
2. `sync()` records the initial balance of `inputCurrency`
3. Permit2 transfers the attacker's worthless token (not `inputCurrency`)
4. `settle()` returns success but doesn't clear the debt
5. `unlock()` detects non-zero delta for `inputCurrency` and reverts

Suggested fix:

Update `BaseSwapRouterPermit2._unlockCallback` as follows:

```
--- a/src/utils/BaseSwapRouterPermit2.sol
+++ b/src/utils/BaseSwapRouterPermit2.sol
@@ -15,6 +15,8 @@ contract BaseSwapRouterPermit2 is BaseSwapRouter {

     constructor(IPoolManager manager, ISignatureTransfer permit2_) BaseSwapRouter(manager,
permit2_) {}

+    error TokenMismatch(address);
+
     function _unlockCallback(bytes calldata callbackData) internal virtual override returns
(bytes memory) {
         unchecked {
             // Decode core swap parameters
@@ -47,6 +49,9 @@ contract BaseSwapRouterPermit2 is BaseSwapRouter {
                 (,,, permitPayload) = abi.decode(callbackData, (BaseData, Currency,
PathKey[], PermitPayload));
             }

+            if (permitPayload.permit.permitted.token != Currency.unwrap(inputCurrency)) {
+                revert TokenMismatch(permitPayload.permit.permitted.token);
+            }
             // Sync pool and handle token pull
             poolManager.sync(inputCurrency);
```

Foundry Test

In `CompliantRouterTest.t.sol`:

```
function test_permitTokenMismatchVulnerability() public {
    // Create a worthless token that attacker controls
    MockERC20 maliciousToken = new MockERC20('MAL', 'MAL', 18);
    maliciousToken.mint(charlie, 1000 ether);

    // Charlie approves the malicious token to Permit2
    vm.prank(charlie);
    maliciousToken.approve(address(permit2), type(uint256).max);

    // Create permit for the MALICIOUS token (not currency0!)
    IAllowanceTransfer.PermitSingle memory ps =
        defaultERC20PermitAllowance(address(maliciousToken), type(uint160).max,
type(uint48).max, 0);
    ps.spender = address(compliantRouter);
    bytes memory sigPermit = getPermitSignature(ps, charliePK, permit2.DOMAIN_SEPARATOR());

    // Setup swap parameters expecting currency0
    uint256 amountIn = 1 ether;
    uint256 amountOutMin = 0;
    uint256 deadline = block.timestamp + 1;

    PathKey[] memory path = new PathKey[](1);
    path[0] = PathKey({
        intermediateCurrency: currency1,
        fee: key.fee,
        tickSpacing: key.tickSpacing,
        hooks: IHooks(levery),
        hookData: abi.encode(charlie)
    });

    // Approve malicious token via permit2
    vm.startPrank(charlie);
    compliantRouter.permit(charlie, ps, sigPermit);

    // Now craft raw swap data with mismatched permit
    BaseData memory bd = BaseData({
            amount: amountIn,
            amountLimit: amountOutMin,
            payer: charlie,
            receiver: charlie,
            flags: SwapFlags.PERMIT2
    });

    PermitPayload memory pp;
    pp.permit.permitted.token = address(maliciousToken); // WRONG TOKEN!

    bytes memory payload = abi.encode(bd, currency0, path, pp);

    // Should revert with 'TokenMismatch(maliciousToken)' if properly validated
    vm.expectRevert(abi.encodeWithSelector(BaseSwapRouterPermit2.TokenMismatch.selector,
address(maliciousToken))); // This SHOULD happen with fix
    // If vulnerable: pulls malicious token but swaps as if it were currency0
    compliantRouter.swap(payload, deadline);
    vm.stopPrank();
}
```

Status

Addressed in commit 367dc9c5136eb187ce22ff90cc8f020c27d62c8c.

A04: Change the loop counters from uint8 to uint256

Severity: Low | Difficulty: Medium | Recommended Action: Fix Code | Addressed by client

Description

The `escapeSpecialCharacters` function uses a uint8 loop counter to iterate over `symbolBytes.length`. If the input symbol string length exceeds `255` bytes, the `uint8` index will overflow (wrap to 0) and never reach `symbolBytes.length`, resulting in an infinite loop and gas exhaustion. This can make `tokenURI` retrieval unusable for positions involving tokens with extremely long symbols.

Recommendation

Use a `uint256` loop counter instead of uint8 for iterating over `symbolBytes.length`, and/or enforce a maximum allowed symbol length before processing to prevent excessively long inputs.

Notes

This finding was identified with the assistance of the Almanax tool.

Auditor Notes:

The compiler's built-in overflow protection will cause a revert when `i++` increments past `255`, preventing an infinite loop.

Recommendation

Change the loop counters from `uint8` to `uint256`.

Foundry test

In `PositionDescriptor.t.sol`:

```
    function test_escapeSpecialCharacters_uint8Overflow() public {
        uint256 stringLength=256;
        // Create a string longer than 255 bytes
        bytes memory longString = new bytes(stringLength);

        // Fill with 'A' characters
        for (uint256 i = 0; i < stringLength; i++) {
            longString[i] = 'A';
        }

        // With the current implementation, this will revert due to uint8 overflow when i
increments from 255 to 256
        wrap.esc(string(longString));
    }
```

Status

Addressed in commit 367dc9c5136eb187ce22ff90cc8f020c27d62c8c

A05: backslash not declared as special character

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The `escapeSpecialCharacters` function in the Descriptor library fails to escape backslash characters (`\`) when preparing strings for JSON encoding. While the function correctly escapes quotes, newlines, tabs, and other control characters, it omits backslashes from the list of special characters that must be escaped.

Recommendation

Add backslash to the list of special characters that must be escaped:

```
function isSpecialCharacter(bytes1 b) private pure returns (bool) {
    return b == '"' || b == "\u000c" || b == "\n" || b == "\r" || b == "\t" || b == "\\";
}
```

Foundry test

In `PositionDescriptor.t.sol` :

```
    function test_escapeSpecialCharacters_missingBackslash() public view {
        // Test that a backslash is NOT escaped (but should be)
        string memory input = 'Test\\Backslash';
        string memory escaped = wrap.esc(input);

        // Currently returns: Test\Backslash (unescaped)
        // Should return: Test\\Backslash (escaped)
        assertEq(escaped, 'Test\\\\Backslash'); // This will PASS (incorrectly)
    }
```

Status

Addressed in commit 5bee1edbb726eae0f75705db0a456cd05514b396

Informative Findings

This section includes observations that are not directly exploitable but highlight areas for improvement in code clarity, maintainability, or best practices. While not critical, addressing these can strengthen the system's overall robustness.

B01: `_poolId` not used consistently across the Levery contract

Severity: Informative | Difficulty: High | Addressed by client

There are still some places where `PoolId.unwrap(key.toId());` is used instead of the `_poolId` private helper.

runtimeverification/_audits_levery-org_levery-contracts/src/Levery.sol

Line 371 to 373 in 970ec76

```
    function _poolId(PoolKey calldata key) private pure returns (bytes32 pid) {
        pid = PoolId.unwrap(key.toId());
    }
```

Recommendation

Use the `_poolId` helper consistently throughout the contract to improve code clarity and maintainability.

Status

Addressed in commit ce793da3a253b4fb26d764f3ff316614f64b7c76.

Client response

Introduce or reuse `_poolId(PoolKey)` and replace direct calls to `PoolId.unwrap(key.toId())` in: `whenPoolNotPaused`, `setPoolBaseFee`, `setPoolOracle`, `removePoolOracle`, `getPoolBaseFee`, `getPoolRequiredRole`, `getPoolOracle`, and role/pausing helpers.

B02: Reading `poolBaseFees` twice in `_adjustSwapFee`

Severity: Informative Difficulty: High Addressed by client

`getPoolBaseFee(key)` is read twice from storage, incurring additional gas costs.

runtimeverification/_audits_levery-org_levery-contracts/src/Levery.sol

Line 385 to 394 in 970ec76

```
    function _adjustSwapFee(PoolKey calldata key, uint256 price0, uint256 price1, bool
zeroForOne, uint24 currentFee)
        internal
        view
        returns (uint24 newSwapFee)
    {
        newSwapFee = getPoolBaseFee(key) != 0 ? getPoolBaseFee(key) : currentFee;
        PoolOracle memory po = poolOracles[PoolId.unwrap(key.toId())];
        if (po.oracle != address(0)) {
            uint256 mp = uint256(getLastOraclePrice(po.oracle, key));
            if (mp == 0) revert OraclePriceZero();
```

Recommendation

Store the pool base fee in a local variable before computing the `newSwapFee`.

Status

Addressed in commit cb9a519be690266ce13c13cf2b03adaa1a041380.

B03: Missing custom error for fee overflow

Severity: Informative Difficulty: High Addressed by client

The fee overflow check `require(fee256 <= type(int128).max && fee256 >= type(int128).min, "Fee overflow")` is duplicated in both `_handleExactInputServiceFee` and `_afterSwap`.

Additionally, custom errors are missing across the `CompliantRouter`, `SoulboundPositionManager`, and `PositionManager` contracts.

Recommendation

The `require` check could be refactored into an if statement, and a new custom error `ServiceFeeOutOfBounds` could be used.

```
error ServiceFeeOutOfBounds(int256);
...
if( fee256 >= type(int128.max) || fee256 <= type(int128.min)) revert
ServiceFeeOutOfBounds(fee256);
```

Status

Addressed in commit 08345c5e559be97234ead8886cf8198dbac3b135

B04: Duplicated logic when computing the serviceFee

Severity: Informative   Difficulty: High   Addressed by client

The service fee calculation logic is repeated in two locations with identical mathematical operations and validation checks. This code duplication creates several maintenance issues that, while not directly exploitable, weakens the codebase’s quality and increases the risk of inconsistent updates.

Location 1:

runtimeverification/_audits_levery-org_levery-contracts/src/Levery.sol

Line 561 to 562 in 970ec76

```
            int256 fee256 = (-inputDelta * int256(serviceFee)) / int256(MAX_PPM);
            require(fee256 <= type(int128).max && fee256 >= type(int128).min, "Fee overflow");
```

Location 2:

runtimeverification/_audits_levery-org_levery-contracts/src/Levery.sol

Line 423 to 425 in 970ec76

```
        int256 fee256 = (-params.amountSpecified * int256(serviceFee)) / int256(MAX_PPM);
        require(fee256 <= type(int128).max && fee256 >= type(int128).min, "Fee overflow");
        int128 fee = int128(fee256);
```

Recommendation

Refactor the logic in a single function, such as:

```
    function _computeServiceFee(uint256 amount) private view returns(uint128){
        uint256 fee = (amount * serviceFee) / MAX_PPM;
        require(fee <= type(int128).max && fee >= type(int128).min, "Fee overflow");
        return uint128(fee);
    }
```

Status

Addressed in commit 2be23e7f96c2cac4cb5a15f61aa09af18b8b701f.

The client added a new function that computes the service fee and returns both the `feeDelta` and the `feeAbs` values.

B05: Inconsistent Return Variable Usage

Severity: Informative Difficulty: High Addressed by client

The function `_computeDynamicFee()` declares a named return variable `adjustedFee` but doesn't use it, instead relying on an explicit return statement:

runtimeverification/_audits_levery-org_levery-contracts/src/Levery.sol

Line 436 to 440 in 970ec76

```
    /// @return adjustedFee Adjusted LP fee in basis points.
    function _computeDynamicFee(PoolKey calldata key, bool zeroForOne) private view
returns (uint24 adjustedFee) {
        (uint256 p0, uint256 p1) = getCurrentPrices(key);
        return _adjustSwapFee(key, p0, p1, zeroForOne, baseFee);
    }
```

Recommendation

Either use the named return variable or remove the unused variable name.

Status

Fixed using the named returned variable in commit 07402ca1a3f559af69a451e8edaf728919ea429f.

B06: Approval functions not restricted in SoulboundPositionManager

Severity: Informative Difficulty: High Addressed by client

Description

The `SoulboundPositionManager` contract implements soulbound NFTs by overriding transfer functions (`transferFrom` , `safeTransferFrom`) so they always revert, preventing token transfers. However, the contract fails to override the approval functions (`approve` , `setApprovalForAll`) inherited from ERC721, allowing users to grant transfer permissions even though transfers will always revert.

This creates an inconsistent state where users can successfully call `approve(spender, tokenId)` and `setApprovalForAll(operator, true)` . These approvals emit events that may misleadingly suggest that transfers are authorized.

Recommendation

Override the approval functions to revert, maintaining consistency with the soulbound design:

```
function approve(address, uint256) public pure override {
   revert("Soulbound: approvals disabled");
}

function setApprovalForAll(address, bool) public pure override {
   revert("Soulbound: approvals disabled");
}
```

Status

Additional tests were added in commit 4c79feaebfe471ecea4ff6a41f1587fdede26f45.

Client Response

`ERC721Permit_v4` 's `approve` and `setApprovalForAll` functions are non-virtual, so they cannot be overridden to revert. Approvals may succeed and emit events, but transfers are already blocked in `SoulboundPositionManager`. We codify this behavior with tests.

Client Findings

During the engagement we confirmed three implementation issues identified by the Levery team and validated their fixes. We are mentioning them below for visibility.

L01 — Native ETH Input Settlement

Fixed in: PR #4

Original issue: `_unlockCallback` forced Permit2 for all inputs, including native ETH.

Fix: When `inputCurrency == ADDRESS_ZERO` , bypass Permit2 and settle ETH directly via `poolManager.sync()` + `settle()` , refunding leftovers. ERC-20 path unchanged (Permit2 + A03 token validation). No breaking changes.

L02 — Decimals-Aware Price Calculation

Fixed in: PR #7

Original issue: Hardcoded `1e18` normalization broke price calculations for mixed-decimal pairs (6/18, etc.).

Fix: Scale by actual decimals (`10^dec0 / 10^dec1`) using `mulDiv` . Reject boundary `sqrtPrice` (MIN/MAX). Now matches oracle price units correctly.

L03 — Symmetric Deviation Metric

Fixed in: PR #9

Original issue: `deviation = |price - oracle| / oracle` over-penalized when oracle < pool.

Fix: Changed to `deviation = |price - oracle| / max(price, oracle)` . Symmetric denominator eliminates bias while preserving direction-aware fee gating. Still capped by `deviationFactor` .