



Security Review For Mancer



Collaborative Audit Prepared For:
Lead Security Expert(s):

Mancer
AV

TessKimy

Date Audited:

August 24 - August 27, 2026

Introduction

Mancer is the non-custodial DEX aggregator and smart-order layer for Robinhood Chain. This audit covers the core contracts responsible for routing swaps and executing limit, stop, and recurring orders while keeping user funds in their wallet until execution. It includes the following features and improvements: Multi-venue swap execution through allowlisted targets and selectors, with measured delivery, fee accounting, and user-defined output protection. Isolated token approvals through a dedicated allowance target and exact, temporary per-leg spending authority. EIP-712 limit, stop, recurring, and one-cancels-the-other orders with maker-controlled cancellation and signed execution bounds. Fixed recurring-order schedules with maker-signed fill windows, skipped-slot accounting, and bounded schedule horizons. Governance hardening through timelocked additions, immediate revocation and pause controls, and improved support for non-standard ERC-20 behavior. A detailed technical explanation of the audited contracts and their security model can be found in the Mancer contracts [repository](#).

Scope

Repository: `blockhash-xyz/mancer-contracts`

Audited Commit: `1968fd5fd20583cabbce09c486f8f0206dfe3396`

Final Commit: `3a5374b4509ab3bea69bfa2a30847b7ce8232711`

Files:

- `src/MancerAllowanceTarget.sol`
- `src/MancerOrders.sol`
- `src/MancerRouter.sol`

Final Commit Hash

`3a5374b4509ab3bea69bfa2a30847b7ce8232711`

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or

related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	3	11

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Predictable Nitro inputs can make `cancelAll()` activate a pre-signed order [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/10>

Summary

`cancelAll()` derives the new maker epoch from chain values described as unknowable before signing. On Nitro, those values can be known or selected early enough to obtain a signature for the exact future epoch.

Vulnerability Detail

The new epoch uses the previous exposed block hash, the current timestamp, maker, and current epoch. On Robinhood's Nitro runtime, the block-number/hash input is sourced from an L1 hash ring and may already be known, while the sequencer controls the admissible L2 timestamp.

A malicious signing interface cooperating with transaction inclusion can:

1. Calculate the epoch that a future `cancelAll()` call will produce.
2. Obtain a maker signature for an order using that future epoch.
3. Confirm that the order is currently rejected with `WrongEpoch`.
4. Cause `cancelAll()` to set the maker epoch to the calculated value.
5. Publish the same order, which is now valid and can be filled.

Impact

The panic button can activate a phishing order that the maker believed would remain invalid, but the loss is bounded by the signed order and its fill schedule.

Code Snippet

```
function cancelAll() external {
    uint64 current = makerEpoch[msg.sender];
    uint64 jump = uint64(
        uint256(
            keccak256(
                abi.encodePacked(
                    blockhash(block.number - 1),
                    block.timestamp,
                    msg.sender,
                )
            )
        )
    );
    makerEpoch[msg.sender] = current + jump;
}
```

```

        current
    )
) >> 224
) + 1;
uint64 next = current + jump;
makerEpoch[msg.sender] = next;
}

```

Relevant code and PoC: `src/MancerOrders.sol:412-426` and `test/MancerOrders.t.sol:606-649`.

Tool Used

Manual Review and Foundry Unit Test

The focused test passed with 1 passed; 0 failed; 0 skipped.

Recommendation

Do not use predictable chain fields as cancellation entropy. Use a maker-supplied fresh random epoch generated after the earlier signatures, or redesign cancellation so that its safety does not depend on an unpredictable next epoch.

Discussion

DemoreXTess

Fix is confirmed

Issue M-2: Missed periods never cost a fill, so a stalled recurring order just stretches instead of skipping slots [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/13>

Summary

The schedule jumps `nextDueAt` over every period that went by, but `fillsDone` still only goes up by one per fill. So nothing is actually skipped, the order still executes all `totalFills`, it just finishes later than the maker signed up for.

Vulnerability Details

The stated behaviour is right there above the advance: missed periods are skipped, never filled in later, because a down executor must not cause a catch up. The code does not do that.

```
// Due when the grid says so. The anchor is startAt (or now, for
// an unanchored first fill); missed periods are SKIPPED, never
// back-filled - a down executor must not cause a lump-sum
// catch-up.
uint40 due = s.nextDueAt != 0 ? s.nextDueAt : o.startAt;
if (block.timestamp < due) revert NotDue();
s.fillsDone += 1;
if (s.fillsDone < o.totalFills) {
    // The next slot is the first grid point strictly after now,
    // computed in O(1): a per-period walk made a long gap cost
    // gas per missed period (a 1-minute order paused a month
    // walked ~43k iterations; a year could not fit in a block),
    // and a signed startAt deep in the past was a free way to
    // burn the executor's simulation budget (review find,
    // 2026-08-20). Same grid, same anchor, same skip semantics.
    uint40 anchor = due == 0 ? uint40(block.timestamp) : due;
    uint40 next = anchor + uint40(((block.timestamp - anchor) / o.everySec + 1) *
        → o.everySec);
    s.nextDueAt = next;
}
```

Half of the promise works. `next` is recomputed as the first grid point strictly after now, so `nextDueAt` really does jump over every slot that passed while the executor was down. The other half is missing. `fillsDone` counts fills that were executed, not periods that elapsed, and it is bumped by exactly one no matter how big the gap was. The skipped slots stay in the order's budget, so they get consumed at the tail of the schedule instead of being dropped.

Concrete run through with a daily recurring order, `everySec = 1 day` and `totalFills = 30`. Five fills happen on time and `nextDueAt` sits at day 5. The executor is then down for ten days. The next fill lands at day 15: `fillsDone` goes from 5 to 6 and `next` becomes day 16. Ten calendar slots disappeared, zero fills disappeared. The order now runs its 30th fill around day 40 instead of day 30 and keeps pulling `amountPerFill` out of the maker's wallet for those extra ten days. Every further outage stretches it again, and the stretch compounds.

Nothing else bounds this. `expiry` is optional and documented as 0 = never, so a maker who leaves it unset has no backstop at all, and even a maker who sets one is relying on a field the schedule logic never consults for this. The maker signed a schedule of 30 daily buys expecting it to be finished in a month, and what they actually get is 30 buys spread over however long the executor decides to take, at prices from a window they never agreed to.

Impact

A recurring order keeps living and keeps pulling the maker's tokens well past the window they signed for, because a missed period costs zero fills instead of one. The documented skip semantics never drop a single fill, so any executor downtime, accidental or deliberate, shifts the schedule out with no bound when `expiry` is 0.

Recommendation

Charge the budget for the slots that went by. Work out how many periods elapsed since `due` and add that count to `fillsDone` (clamped at `totalFills`) instead of always adding one, so a missed period actually costs a fill and the order still ends when the maker expected it to.

Discussion

DemoreXTess

Fix is confirmed

Issue M-3: No upper bound on fill lateness, and a late fill can be followed by the next one a second later [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/14>

Summary

Nothing limits how late a fill may be, and `nextDueAt` is derived from the old grid point rather than from when the fill actually happened. Fill one second before a grid point and the next fill is due one second later, so two fills land back to back on an order whose minimum period is 60 seconds.

Vulnerability Details

```
uint40 due = s.nextDueAt != 0 ? s.nextDueAt : o.startAt;
if (block.timestamp < due) revert NotDue();
s.fillsDone += 1;
if (s.fillsDone < o.totalFills) {
    uint40 anchor = due == 0 ? uint40(block.timestamp) : due;
    uint40 next = anchor + uint40(((block.timestamp - anchor) / o.everySec + 1) *
    → o.everySec);
    s.nextDueAt = next;
}
```

Now look at what `next` is anchored to. `anchor` is the previous due point, not the moment this fill actually settled, so the spacing of the next slot is measured from the grid and not from the fill. Take the gap between now and the anchor. When that gap is one second short of a full `everySec` period, the integer division gives 0 and `next` comes out as `anchor + everySec`, which is exactly one second after the fill that just happened. So the executor settles the slot one second before the grid point, waits for the next block, and fills again straight away. Two fills one second apart on an order where `MIN_PERIOD` is 60 seconds and the maker signed a period of at least that.

The whole point of splitting an order into `N` fills every `everySec` seconds is that the maker gets `N` separate price points spread over time. Two fills a second apart are one price point with double the size, so the maker eats the price impact of the combined amount at a single instant and gets no averaging out of the pair.

Impact

The executor can settle a recurring fill arbitrarily late and can pack two fills into consecutive blocks, so the maker loses the period spacing they signed for and takes both

fills at one price and one moment of the executor's choosing instead of spread across the schedule.

Recommendation

Add a lateness window so a fill reverts when `block.timestamp` is too far past due, and derive the next slot from the fill itself, for example by requiring `nextDueAt` to be at least `block.timestamp + everySec`, so two fills can never sit closer than a full period.

Discussion

DemoreXTess

Fix is confirmed, however fix method introduced a new issue:

```
if (block.timestamp < due) revert NotDue();
if (o.totalFills > 1 && s.fillsDone != 0 && block.timestamp <
    ↪ uint256(s.lastSuccessfulAt) + o.everySec) {
    revert NotDue();
}
```

`due` is calculated with `anchor + slots * everySec` formula however, second due check operates based on the last execution time. This condition may force some fills to skip:

1. Everysec = 10 minutes , 50 total fills
2. First fill executed at `t`.
3. Second fill executed 30 seconds later than expected = `t+10minutes+30 seconds`

If every fill executed with 30 seconds delay, total delay will be 10 minutes after 20 fills. At this points due will point out `t + 200 minutes` but `lastSuccessfulAt` will point out `t + 200 minutes + 10 minutes`.

This is medium severity issue because it actually makes `if (block.timestamp < due) revert NotDue();` is a deadcode because only last execution time will matter here. We can't know when this schedules call series will end and when intermediate fills will be executed because we can't give a guarantee that execution will be applied on exact time.

For instance, a maker wanted to make a trade everyday on 6AM, for the next 3 months. Correct order setup would be :

```
everySec = 24 hours
totalFills = 90
startAt = next 6AM timestamp
```

This setup won't work if delay happens and some fills can be executed even after 4 hours from expected hour in rare conditions (depends on executor)

I think allowing maker to specify maximum delay in order and reverting it based on that threshold is better idea.

mikeahirsch

<https://github.com/blockhash-xyz/mancer-contracts/pull/12>

DemoreXTess

Fix is partial. It doesn't solve the problem entirely. It solves unintended execution timelines but it still shifts the time.

You need to remove `s.lastSuccessfulAt` and `cooldownAt` variables. By doing that maker will be able to schedule a periodic fill with exact time with extra allowed time window.

mikeahirsch

<https://github.com/blockhash-xyz/mancer-contracts/pull/13>

DemoreXTess

Fix is confirmed

Issue L-1: Unsafe fee-recipient aliases can refund or strand protocol fees [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/4>

Summary

The owner can configure the Router, Orders contract, or AllowanceTarget as the fee recipient. Trades still succeed, but the fee can be refunded to the user or maker, or become stuck instead of reaching the intended fee converter.

Vulnerability Detail

`setFeeRecipient()` rejects only the zero address. It does not reject protocol-owned addresses whose behavior conflicts with fee collection.

For example:

- If the Router is the recipient, the fee remains in the Router and can be returned by the final sweep.
- If Orders is the recipient, Orders treats the received fee as new residue and forwards it to the maker.
- If AllowanceTarget is the recipient, the tokens have no recovery path through its limited interface.

This requires an owner or deployment mistake. An unprivileged user cannot change the recipient.

Impact

The protocol can lose up to all fees collected while the unsafe configuration is active. This is an operational configuration issue, not a public wallet-drain vulnerability.

Code Snippet

```
function setFeeRecipient(address feeRecipient_) external onlyOwner {
    if (feeRecipient_ == address(0)) revert ZeroAddress();
    feeRecipient = feeRecipient_;
}

uint256 feeSend = (a.produced * p.feeBps) / 10_000;
if (feeSend > 0) {
```

```
a.feeAmount = _transferMeasured(tout, feeR, feeSend);  
}
```

Relevant code: `src/MancerRouter.sol:315-359,597-605,761-764`.

Tool Used

Manual Review

Recommendation

Reject the Router, Orders, AllowanceTarget, WETH, and other unsafe protocol aliases in the constructor and setter.

Discussion

DemoreXTess

Fix is confirmed

Issue L-2: Orders can be created with a start time after their expiry [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/5>

Summary

Order validation accepts an order whose first possible execution time is later than its expiry. Such an order can be signed successfully but can never be filled.

Vulnerability Detail

The static validator checks the individual order fields but does not require `startAt <= expiry` when an expiry is present.

For example, an order with `startAt = 1,001` and `expiry = 1,000` behaves as follows:

- Before timestamp 1,001, the order is not due.
- From timestamp 1,001 onward, the order is expired.

There is no timestamp at which both conditions pass.

Impact

The maker receives an unusable signed order, and executors may waste effort attempting to simulate or submit it. No funds are transferred because every fill attempt reverts.

Code Snippet

```
// _validate() does not check:
// o.expiry == 0 || o.startAt <= o.expiry

if (o.expiry != 0 && block.timestamp > o.expiry) revert Expired();

uint256 due = s.nextDueAt == 0 ? o.startAt : s.nextDueAt;
if (block.timestamp < due) revert NotDue(due);
```

Relevant code: `src/MancerOrders.sol:324-340,359-380`.

Tool Used

Manual Review

Recommendation

Reject impossible schedules on-chain and in the order builder:

```
if (o.expiry != 0 && o.startAt > o.expiry) revert BadSchedule();
```

Discussion

DemoreXTess

Fix is confirmed

Issue L-3: Aggregate accounting can hide a wrong-recipient or wrong-asset leg [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/6>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Mancer validates output only at the complete-plan level. It does not verify that each external leg produced its declared token or sent its proceeds to Mancer.

Vulnerability Detail

Mancer checks the target and the first four payload bytes, then sends the opaque payload to the external router. `Leg.tokenOut` is not trusted for accounting; it is only added to the final sweep list.

After every leg finishes, Mancer measures a single increase in the plan's final token. This allows the following example:

1. Leg 1 sends its output to another receiver or produces undeclared token X.
2. Leg 2 supplies enough of the declared final token B.
3. The aggregate B floor passes.
4. The redirected output or undeclared X is not recovered.

Impact

A direct caller normally redirects its own money. In Orders, the route is supplied by a trusted executor and the maker still receives the signed aggregate minimum. The confirmed issue is therefore a per-leg conservation and integration-correctness failure.

Code Snippet

```
// Leg.tokenOut is only a sweep hint.
struct Leg {
    address target;
    bytes payload;
    address tokenIn;
    address tokenOut;
    uint256 amountIn;
    FundMode fund;
}
```

```
for (uint256 i = 0; i < p.legs.length; i++) {
    _runLeg(p.legs[i], feeR);
}

a.produced = IERC20(tout).balanceOf(address(this)) - a.outStart;
a.delivered = _deliverMeasured(p.tokenOut, p.recipient, tout, a.netOut);
if (a.delivered < p.minAmountOut) revert TooLittleReceived();
```

Relevant code: `src/MancerRouter.sol:227-249,333-398,422-475,597-633` and `src/MancerOrders.sol:202-245`.

Tool Used

Manual Review

Recommendation

Use target-specific adapters or payload decoders that bind each leg's input token, output token, recipient, amount, and completion semantics.

Discussion

mikeahirsch

The maker still receives the amount they signed for, and cannot lose more input than `amountPerFill`. #17 also ensures their complete token-B receipt respects the signed maximum. What can be lost is favorable surplus or an undeclared intermediate token due to a bad route. Enforcing every leg on-chain would require understanding each venue's opaque calldata and would undermine the generic aggregator design. The right controls are off-chain payload validation and selective typed adapters for risky venues.

Issue L-4: The order signature does not bind the fee asset or fee basis [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/7>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

An order signature limits only the numerical fee rate. It does not specify whether the fee is taken from input, output, or an intermediate token.

Vulnerability Detail

Order hashes `maxFeeBps`, but the unsigned `FillPlan` contains `feeMode`, `feeAsset`, `feeAfterLeg`, and `minFeeBasis`.

The same signed order can therefore be filled using:

- An input fee charged over received input.
- An output fee charged over produced output.
- A mid-route fee charged in an executor-selected intermediate token.

The numerical basis-point cap is enforced which means that the signed authorization does not define what asset or economic basis those basis points apply to, which contradicts the documentation.

Impact

A trusted executor or changed API fee schedule can alter the fee economics of an already signed recurring order which represents a contradiction of the documentation.

Code Snippet

```
struct Order {
    // ...
    uint16 maxFeeBps;
}

struct FillPlan {
    uint16 feeBps;
    MancRouter.FeeMode feeMode;
    address feeAsset;
    uint16 feeAfterLeg;
```

```
uint256 minFeeBasis;  
// ...  
}  
  
if (feeBps > o.maxFeeBps) revert FeeOverCap();
```

Relevant code: `src/MancerOrders.sol:126-166,208-239,291-301` and `src/MancerRouter.sol:306-371`.

Tool Used

Manual Review

Recommendation

Include the fee mode, asset, checkpoint, and basis in the signed order. An alternative is to sign a maximum absolute fee denominated in a clearly defined asset.

Discussion

mikeahirsch

Only maxFeeBps is signed; fee placement and asset are executor-selected. Signing all route-dependent fee fields would make long-lived orders brittle. Will update the docs.

Issue L-5: Exact selector revocation does not override a wildcard [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/8>

Summary

Removing one selector from a target does not block it when ANY_SELECTOR remains enabled for that target.

Vulnerability Detail

Runtime permission is calculated as `exactSelector || ANY_SELECTOR`. `setCalls(target, [selector], false)` clears only the exact selector entry and emits a revocation event. It does not create a negative override.

An operator may therefore believe a dangerous function has been disabled while `callAllowed()` and actual execution continue to allow it through the wildcard.

Impact

Narrow incident response can be ineffective. Operators must remove the complete wildcard or disable the target, which also removes unrelated functionality.

Code Snippet

```
if (!isCall[l.target][sel] && !isCall[l.target][ANY_SELECTOR]) {
    revert CallNotAllowed();
}

function setCalls(address target, bytes4[] calldata selectors, bool allowed)
    external
    onlyOwner
{
    for (uint256 i = 0; i < selectors.length; i++) {
        isCall[target][selectors[i]] = allowed;
    }
}
```

Relevant code: `src/MancerRouter.sol:439-458,690-733`.

Tool Used

Manual Review

Recommendation

Add a per-target negative selector mapping that is checked before the wildcard or reject an exact revocation while the wildcard remains active and require operators to revoke the wildcard explicitly

Discussion

DemoreXTess

Fix is confirmed

Issue L-6: The selector denylist omits standard asset-moving functions [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/9>

Summary

The Router's defense-in-depth denylist blocks twelve asset-moving selectors but omits several canonical functions with similar authority semantics.

Vulnerability Detail

The denylist is intended to catch mistakes when a target is enabled through a wildcard. However, it does not include examples such as:

- Permit2 single and batch `transferFrom`.
- ERC-777 `operatorBurn`.
- ERC-3525 value-transfer overloads.
- ERC-7540 `requestRedeem`.

Impact

The omission increases the consequence of a future wildcard target that recognizes Mancer as a spender, operator, or privileged caller.

Code Snippet

```
if (
    sel == SEL_TRANSFER_FROM
    || sel == SEL_OPERATOR_SEND
    || sel == SEL_4626_WITHDRAW
    || sel == SEL_4626_REDEEM
    || sel == SEL_721_SAFE_TRANSFER_FROM
    || sel == SEL_1155_SAFE_TRANSFER_FROM
    || sel == SEL_6909_TRANSFER_FROM
) revert PayloadRefused();
```

The listed checks do not include the Permit2, ERC-777 burn, ERC-3525, or ERC-7540 selectors described above. Relevant code: `src/MancerRouter.sol:157-182,439-458`.

Tool Used

Manual Review

Recommendation

Add the high-signal omitted selectors relevant to supported integrations. Keep the list and its tests table-driven, document that it is defense in depth, and avoid wildcards for authority-bearing targets.

Discussion

DemoreXTess

Fix is confirmed

Issue L-7: Forced ETH can temporarily disable UP33-CL routes [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/11>

Summary

An attacker can force one wei of ETH into UP33-CL. Its next Mancer-routed swap attempts to refund that ETH to Mancer, but Mancer accepts raw ETH only from WETH, causing the complete route to revert.

Vulnerability Detail

The admitted UP33-CL `exactInputSingle` function unconditionally calls `refundETH()` after the swap. That helper sends the external router's complete ETH balance to `msg.sender`, which is Mancer.

Mancer's receiver rejects the payment because the sender is UP33-CL rather than WETH. The swap and Orders state then roll back atomically.

Anyone can later clear the condition by calling UP33-CL's public `refund` function from an ETH-receiving address, but an attacker can force-send dust again or race a prior simulation.

Impact

Affected direct swaps and recurring fills can revert and waste gas, if the UP33-CL route is present.

Code Snippet

External UP33-CL flow:

```
amountOut = exactInputInternal(params);
require(amountOut >= params.amountOutMinimum);
refundETH(); // refunds the router's complete ETH balance to Mancer
```

Mancer receiver:

```
receive() external payable {
    if (msg.sender != weth) revert NotWeth();
}
```

Relevant local code: `src/MancerRouter.sol:520-525,792-796`.

Tool Used

Manual Review

Recommendation

Use an adapter that can receive the external refund, wrap it into WETH, and return it under Mancer's declared asset rules. Otherwise, exclude UP33-CL whenever its ETH balance is nonzero or let UP33-CL as one trusted sender to the Router beside WETH.

Discussion

DemoreXTess

Fix is confirmed

Issue L-8: Resetting the venue allowance to zero bricks swaps for tokens that reject zero value approvals [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-08-mancer-aug-24th-2026/issues/12>

Summary

After the venue call, `_callApproved` wipes the leftover allowance with `forceApprove(target, 0)`. Some ERC20s revert on an approve with a zero amount, so for those tokens the cleanup reverts and the whole swap dies even though the trade itself went fine.

Vulnerability Details

The Approve funding mode grants the venue an exact allowance, runs the payload, then zeroes the allowance again:

```
function _callApproved(address target, bytes calldata payload, address tin, uint256
↳ swapIn) internal {
    IERC20(tin).forceApprove(target, swapIn);
    (bool ok, bytes memory reason) = target.call(payload);
    if (!ok) revert SwapCallFailed(reason);
    IERC20(tin).forceApprove(target, 0);
}
```

The first two steps are fine. The problem is the last one. There are ERC20 implementations out there that explicitly reject `approve(spender, 0)`, the classic example being BNB, whose approve requires a non zero value. `forceApprove` does not save you here either. It first tries `approve(spender, 0)`, and if that fails it retries by setting the allowance to zero and then to the requested value, which is zero again. Every branch ends up calling approve with zero, so all of them revert and `forceApprove` bubbles up `SafeERC20FailedOperation`.

The Permit2 lane has the same shape. `_executePermit2` finishes with `IERC20(tin).forceApprove(p2, 0)`, which fails for the same reason on the same tokens.

Impact

Swaps in Approve mode and Permit2 mode revert for any token whose approve rejects a zero amount, so those tokens are simply untradeable through the router. Not a fund loss, just a permanent functional dead end for that class of token.

Recommendation

Wrap the zero reset in a try/catch, or skip it when the remaining allowance is already zero, so a token that refuses a zero approval does not take the swap down with it. Apply the same fix to the Permit2 lane.

Discussion

DemoreXTess

Fix is confirmed

Issue L-9: Protocol fee always rounds down, so the router leaks fee on every swap and charges nothing on small ones [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/15>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Every fee in `execute` is plain integer division, so it always truncates in the user's favor and never in the protocol's. On top of the dust lost per trade, any trade small enough that `amount * feeBps < 10_000` pays literally zero fee.

Vulnerability Details

There are three places a fee gets taken, one per `FeeMode`, and all three use the same `(amount * p.feeBps) / 10_000` pattern:

```
if (p.feeMode == FeeMode.Input) {
    uint256 feeSend = (a.received * p.feeBps) / 10_000;
    if (feeSend > 0) a.feeAmount = _transferMeasured(tin, feeR, feeSend);
}

uint256 feeSend = (basis * p.feeBps) / 10_000;
if (feeSend > 0) a.feeAmount = _transferMeasured(p.feeAsset, feeR, feeSend);

if (p.feeMode == FeeMode.Output) {
    // The user's net comes off what LEAVES the router (feeSend);
    // the event's feeAmount is what ARRIVES at the recipient.
    uint256 feeSend = (a.produced * p.feeBps) / 10_000;
    a.netOut = a.produced - feeSend;
    if (feeSend > 0) a.feeAmount = _transferMeasured(tout, feeR, feeSend);
}
```

Solidity division truncates toward zero, so whatever remainder the fee has just gets dropped and stays with the user. In Input mode it stays in the amount that goes on to trade, in Mid mode it stays in the cash the later legs spend, and in Output mode you can see it directly because the user's net is the produced amount less whatever the fee failed to take. It is never symmetric. There is no path where the rounding lands on the protocol's side.

Impact

The protocol quietly gives up fee revenue on every trade, and trades under the truncation cutoff pay no fee at all.

Recommendation

Round the protocol fee up instead of down, something like $(\text{amount} * \text{p.feeBps} + 9_999) / 10_000$, and skip the fee only when the basis itself is zero rather than when the rounded fee is zero.

Discussion

mikeahirsch

The arithmetic is correct, but this is intentional user-favoring integer rounding rather than a vulnerability. The maximum shortfall is less than one smallest unit of the fee asset per transaction. Rounding up would introduce a one-unit minimum fee that can greatly exceed the signed basis-point cap on small amounts. I'm retaining floor rounding; minimum trade sizes can be enforced separately if needed.

Issue L-10: The 60 second Permit2 expiration does not bound anything [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-08-mancer-aug-24th-2026/issues/16>

Summary

`_executePermit2` grants the venue a Permit2 allowance that expires 60 seconds after `block.timestamp`, and the natspec above it states that window as the "never after this trade" bound. It is not a bound at all: everything in the transaction sees one timestamp, and the timestamp itself is picked by whoever submits the transaction.

Vulnerability Details

Here is the whole Permit2 lane:

```
function _executePermit2(address target, bytes calldata payload, address tin,
    ↪ uint256 inAmt) internal {
    address p2 = permit2;
    if (p2 == address(0)) revert Permit2Unset();
    IERC20(tin).forceApprove(p2, inAmt);
    IAllowanceTransferMinimal(p2).approve(tin, target, uint160(inAmt),
    ↪ uint48(block.timestamp + 60));
    (bool ok, bytes memory reason) = target.call(payload);
    if (!ok) revert SwapCallFailed(reason);
    IAllowanceTransferMinimal(p2).approve(tin, target, 0, 0);
    IERC20(tin).forceApprove(p2, 0);
}
```

The grant is made, the venue is called, the grant is revoked. All of that happens inside one transaction, so every read of `block.timestamp` in that span returns the same number. Permit2 only rejects a pull when `block.timestamp > expiration`, and with the expiration set 60 seconds ahead of the current block that check cannot fail during `target.call(payload)`. It would also not fail if the value were 1 second, or 0, or an hour. So during the only moment the allowance is meant to be usable, the window contributes exactly nothing.

Impact

No funds are at risk, but the advertised time bound is decorative.

Recommendation

Set the expiration time as a parameter supplied by the executor.

Discussion

DemoreXTess

Fix is confirmed

Issue L-11: maxOutPerFill ignores output refunded by MancerOrders, allowing out-of-range fills [RESOLVED]

Source: <https://github.com/sherlock-audit2/2026-08-mancer-aug-24th-2026/issues/17>

Summary

MancerOrders.fill() checks maxOutPerFill before refunding tokens held by MancerOrders.

As a result, output sent to MancerOrders during execution is not included in the cap check, even though it is refunded to the maker immediately afterward. The maker can therefore receive more tokenOut than their signed maximum while the order is still marked as filled.

Vulnerability Detail

The router measures only the tokenOut that reaches MancerRouter:

```
a.produced = IERC20(tout).balanceOf(address(this)) - a.outStart;
a.netOut = a.produced;

a.delivered =
    _deliverMeasured(p.tokenOut, p.recipient, tout, a.netOut);
```

MancerOrders then checks the returned amount against the maker's cap:

```
amountOut = router.execute(...);

if (
    o.maxOutPerFill != 0 &&
    amountOut > o.maxOutPerFill
) revert OutOverCap();
```

However, the executor controls the venue calldata used by each leg. A supported venue can send some output directly to MancerOrders.

Because tokenOut is included in the residue list, that additional output is later refunded to the maker:

```
n = _noteToken(toks, n, o.tokenOut);

for (uint256 i = 0; i < nToks; i++) {
    _refund(toks[i], o.maker, keeps[i]);
}
```

The refund happens after the cap check:

```
function _refund(address token, address maker, uint256 keep)
    internal
{
    uint256 bal = IERC20(token).balanceOf(address(this));

    if (bal > keep) {
        IERC20(token).safeTransfer(maker, bal - keep);
    }
}
```

For example, consider an order with:

```
amountPerFill:    1,000 token A
minOutPerFill:     90 token B
maxOutPerFill:     110 token B
```

The executor can build the following route:

```
550 token A → 110 token B sent to MancerOrders
450 token A → 90 token B sent to MancerRouter
```

The result is:

1. MancerRouter measures and delivers 90 token B.
2. MancerOrders receives amountOut = 90.
3. The maximum check passes because $90 \leq 110$.
4. `_refund()` transfers the additional 110 token B from MancerOrders to the maker.
5. The maker receives 200 token B in total.
6. The order is still recorded as successfully filled.

This issue does not depend on output being stolen or sent to an arbitrary third party. All the relevant output ultimately reaches the maker. The problem is specifically that the signed maximum is checked against an incomplete amount.

Impact

A malicious or compromised authorized executor can execute an order outside the maker's signed price range.

The maker spends the complete input amount, while the fill count, schedule, and any OCO state are advanced even though the maker receives more than `maxOutPerFill`.

The maker receives additional output, however, the execution still violates an explicitly signed order condition and can consume an order or scheduled fill at a price the maker did not authorize.

Code Snippet

Affected code:

```
amountOut = router.execute(...);

// Only the amount delivered through MancerRouter is checked.
if (
    o.maxOutPerFill != 0 &&
    amountOut > o.maxOutPerFill
) revert OutOverCap();

// Additional tokenOut held by MancerOrders reaches the maker afterward.
for (uint256 i = 0; i < nToks; i++) {
    _refund(toks[i], o.maker, keeps[i]);
}
```

Location: MancerOrders.sol, inside fill().

Tool Used

Manual Review

Recommendation

Measure the maker's complete tokenOut balance increase across the entire fill, including refunds, and enforce the signed cap after all refunds have been completed:

```
uint256 makerOutBefore =
    IERC20(o.tokenOut).balanceOf(o.maker);

router.execute(...);

for (uint256 i = 0; i < nToks; i++) {
    _refund(toks[i], o.maker, keeps[i]);
}

uint256 actualMakerOut =
    IERC20(o.tokenOut).balanceOf(o.maker) -
    makerOutBefore;

if (
    o.maxOutPerFill != 0 &&
    actualMakerOut > o.maxOutPerFill
) revert OutOverCap();

amountOut = actualMakerOut;
```

This ensures `maxOutPerFill` is enforced against everything the maker actually receives during the fill, not only the portion routed through `MancerRouter`.

Discussion

DemoreXTess

Fix is confirmed

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.