Stryke
Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Devtooligan, Security Researcher

August 26, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Stryke is a revolutionary DeFi derivatives protocol that transforms traditional liquidity provision into high-yield option writing opportunities. Stryke allows users to deploy perpetuals and options for any token pair with up to 100x leverage while eliminating the risk of price-based liquidations. The protocol supercharges yields for Uniswap V3/V4 LPs, Pendle PTs, and LRTs by enabling them to write options and earn premium income on top of standard trading fees, achieving higher returns for the same liquidity. Through its permissionless design, Stryke instantly supports exotic tokens and memecoins, while its omni-chain architecture unifies liquidity across multiple networks. By solving the core problems of high margin requirements, counterparty risk, and suboptimal LP yields, Stryke stands as the complete, risk-minimized derivatives solution that bridges the gap between traditional finance complexity and DeFi innovation.

From Aug 5th to Aug 12th the Cantina team conducted a review of marginzero-xyz on commit hash e3c913ad. The team identified a total of **28** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	0	1
Medium Risk	8	4	4
Low Risk	6	3	3
Gas Optimizations	0	0	0
Informational	13	5	8
Total	**28**	**12**	**16**

3 Findings

3.1 High Risk

3.1.1 In-range settlement refunds are computed with boundary formulas, leaking free tokens to the settler

Severity: High Risk

Context: OptionMarketOTMFE.sol#L538-L564

Description: Inside `settleOption`'s `else` branch (in-range settlement), the contract computes "required" token amounts using boundary helpers:

```
uint256 actualAmount0 = LiquidityAmounts.getAmount0ForLiquidity(
    opTick.tickLower.getSqrtRatioAtTick(),
    opTick.tickUpper.getSqrtRatioAtTick(),
    uint128(liquidityToSettle)
);

uint256 actualAmount1 = LiquidityAmounts.getAmount1ForLiquidity(
    opTick.tickLower.getSqrtRatioAtTick(),
    opTick.tickUpper.getSqrtRatioAtTick(),
    uint128(liquidityToSettle)
);
```

Those are the amounts needed at the range boundaries (as if you provided only one token at sqrtA or sqrtB). They are not the amounts required at the current price sqrtP. For a position with range `[sqrtA, sqrtB]`, liquidity L, and current price sqrtP strictly inside the range (sqrtA < sqrtP < sqrtB), the correct in-range amounts are:

```
// Correct, at current price
amount0_current = L * (sqrtB - sqrtP) / (sqrtP * sqrtB);
amount1_current = L * (sqrtP - sqrtA);

// Boundary-only helpers (upper bounds)
getAmount0ForLiquidity(sqrtA, sqrtB, L) = L * (sqrtB - sqrtA) / (sqrtA * sqrtB);
getAmount1ForLiquidity(sqrtA, sqrtB, L) = L * (sqrtB - sqrtA);
```

It always holds that:

- `getAmount0ForLiquidity(sqrtA, sqrtB, L)` > `amount0_current`.
- `getAmount1ForLiquidity(sqrtA, sqrtB, L)` > `amount1_current`.

Despite this, the settlement code treats `actualAmount0/1` as if they were the actual required amounts and refunds the difference to the settler:

```
// Example path (call option, in-range, expired):
ac.assetToUse.safeIncreaseAllowance(address(positionManager), amount0);     // amount0_current
ac.assetToGet.safeIncreaseAllowance(address(positionManager), amount1);     // amount1_current

// Settler supplies the "other side" at current price:
ac.assetToGet.safeTransferFrom(msg.sender, address(this), amount1);         // amount1_current

// Refund the "excess" in call asset, computed using boundary formula:
ac.assetToUse.safeTransfer(msg.sender, actualAmount0 - amount0);            // > 0
```

Because `actualAmount0 > amount0` (and symmetrically for `token1`), the settler receives free tokens every time settlement happens in-range. This loss is borne by LPs (the option liquidity).

Proof of Concept: *See gist 58f3a64a*

`test_smallLiqRange` (193990, 194000, settled at 193991) yields 1.52 USD profit for the settler, widening the band in `test_longLiqRange` (193990, 194600 settled at 193991), pushes settler's profit to 112.65 USD, with the same settlement tick. This demonstrates the leak is caused by refunding `boundary - current` in the in-range-expired branch.

- Narrow band (193990-194000) settled at 193991 = $1.52 profit.
- Same settlement tick but wide band (193990-194600) = $112.65 profit.

Recommendation: Ensure that only the LPer (instead of a privileged protocol account) can settle in this case so the actual profit is given directly to him.

Stryke: Acknowledged. This is an expected desired outcome from the contract. This is meant to happen in emergency situations when the range is too wide and we need to still settle so, that LP can withdraw. The expected contract flow will not lead to this scenario, as we use tight range for the options mint, and we wait for the price to move out of the range when we settle. Since the range is tight, the price moves out of the defined tick boundary and we settle. However, in the case in future where a large range are supposed to be used, and the LP needs to take an exit, this function allows us to settle, and let LP take the exit, where we donate the profit to the LP.

Cantina Managed: Acknowledged.

3.2 Medium Risk

3.2.1 Users can exercise other user's options

Severity: Medium Risk

Context: ExerciseOptionFirewall.sol#L71-L84

Description: In `exerciseOption` the contract checks authorization and performs range/expiry validations against the top-level `optionId`, but it later executes the settlement using `settleParams.optionId`. There is no assertion that these two IDs are equal.

Simplified flow:

```
function exerciseOption(
    IOptionMarketOTMFE market,
    uint256 optionId, // <- checks use this
    IOptionMarketOTMFE.SettleOptionParams memory settleParams, // <- but execute with this
    RangeCheckData[] calldata rangeCheckData,
    Signature[] calldata signature
) external {
    // Ownership/range checks are tied to `optionId`
    require(market.ownerOf(optionId) == msg.sender, "not owner");
    _verifyRangeAndTime(optionId, rangeCheckData, signature);

    // ... later
    market.settleOption(settleParams); // uses settleParams.optionId internally
}
```

Because `settleParams.optionId` is user-controlled and never compared to `optionId`, a malicious caller who owns option A can pass `settleParams.optionId = B` (a victim's option). The firewall's checks (ownership, tick bounds, time window) are evaluated for A, but the settlement is carried out for B.

Even though proceeds are paid to `market.ownerOf(settleParams.optionId)` (the victim), this is still a serious authorization bypass:

- An attacker can force-exercise another user's option early (or when ineligible per policy), destroying its time value and optionality.
- If settlement marks the option as settled/closed, the victim loses the right to exercise later. In OTM or unfavorable conditions this can realize zero or suboptimal payout, causing direct economic loss to the holder.

Therefore, holders can have their options prematurely closed without consent, losing the premium's remaining time value and potential future intrinsic value.

Recommendation: Consider binding the IDs, by rejecting any call where the two IDs differ:

```
require(settleParams.optionId == optionId, "optionId mismatch");
```

Stryke: Fixed in commit bdd8afc.

Cantina Managed: Fix verified.

3.2.2 DoS near expiry: Division by zero in pricing when `expiry - block.timestamp < 864`

Severity: Medium Risk

Context: OptionMarketOTMFE.sol#L262-L266

Description: `OptionPricingLinearV2` buckets time to expiry in 864-second "days":

```
function _getOptionPrice(OptionPriceParams memory _params) internal view returns (uint256) {
    // floor division in seconds -> buckets of 864s
    uint256 timeToExpiry = _params.expiry.sub(block.timestamp).div(864);
    // ... timeToExpiry is then used in pricing math
}
```

When the remaining time is less than 864 seconds, the floor division sets `timeToExpiry` to 0. Downstream math divides by, or otherwise relies on, this bucketed value and reverts with division by zero during `getOptionPrice` (observable in your call trace just before expiry).

Because the mint/expiry windowing currently allows mints up to the boundary (or only blocks the last 600s), users can submit mint transactions in the final ~14m24s of an epoch that always revert, wasting gas. This is a denial-of-service at the end of every epoch for TTL=1 day. If a market config ever sets ttl < 864, pricing will always revert.

Example (TTL = 86400s):

- Expiry - now = 863 -> timeToExpiry = 863 / 864 = 0.
- Subsequent pricing uses timeToExpiry → division by zero revert.

This contradicts the intended "risk buffer" semantics: users appear allowed to mint right up to expiry, but those mints are unfillable due to pricing failure.

Recommendation: Consider enforcing a pre-expiry buffer ≥ BUCKET so calls cannot enter the danger window.

```
uint256 start = ttlStartTime[_params.ttl];
uint256 into = (block.timestamp - start) % _params.ttl;
uint256 timeUntilExpiry = _params.ttl - into;

// Pick a buffer strictly  864s
uint256 constant BUFFER_BEFORE_END = 900; // e.g., 15 minutes
if (timeUntilExpiry <= BUFFER_BEFORE_END) revert InBufferBeforeEnd();
```

If you keep the 864-second bucketing, `BUFFER_BEFORE_END` must be ≥ 864, otherwise users will still be allowed to attempt mints in a period where `timeToExpiry == 0` and pricing reverts.

Stryke: Acknowledged. We have updated the docs and frontend about the last 864 seconds, which is acceptable for us and our users.

Cantina Managed: Acknowledged.

3.2.3 Premiums & AMM fees are custody-routed to feeReceiver with no on-chain entitlement for LPers

Severity: Medium Risk

Context: V3BaseHandler.sol#L442-L472

Description: LPers mint ERC-6909 "shares" and are expected to earn option premiums paid at mint time and swap fees accrued by the Uniswap V3 position. However, neither stream is credited to LP token holders on-chain. Instead, both are sent to a single `feeReceiver` address:

```
// During option mint: premium sent straight to feeReceiver
if (_params.amount0 > 0) {
    IERC20(tki.token0).safeTransferFrom(msg.sender, feeReceiver, _params.amount0);
}
if (_params.amount1 > 0) {
    IERC20(tki.token1).safeTransferFrom(msg.sender, feeReceiver, _params.amount1);
}

// During fee collection: AMM fees also sent to feeReceiver
_pool.collect(feeReceiver, _tickLower, _tickUpper, tki.tokensOwed0, tki.tokensOwed1);
```

There is no on-chain accounting that attributes these proceeds to a specific ERC-6909 tokenId or to individual LPs. The project clarified that distributions are performed off-chain via "Merkle claims" on epoch schedules. This introduces a custodial/availability dependency: LPs' rewards exist only as an off-chain promise until an operator publishes epochs and funds a distributor.

Recommendation: Consider implement a trustless, on-chain attribution by routing premiums/fees into a non-custodial accounting flow credited per `tokenId`.

Stryke: Acknowledged. We collect all the fees on our feeReceiver address, and we use our offchain method to distribute these rewards. So, all the LPs receive the AMM fees, AMM rewards and premiums as one claim, via our offchain merkl and distribution system.

Cantina Managed: Acknowledged.

3.2.4 Partial reliance on slot0 spot price can enable single-block price manipulation

Severity: Medium Risk

Context: OptionMarketOTMFE.sol#L659-L665, PoolSpotPrice.sol#L9-L17

Description: Across the codebase the protocol reads the Uniswap v3 pool's instantaneous spot from `pool.slot0()` and uses it as if it were a robust price oracle. Concretely:

- Premium/quote calculation: `PoolSpotPrice.getSpotPrice(pool, base, decimals)` pulls `slot0().sqrtPriceX96/slot0().tick` to compute `currentPrice`, which feeds `getOptionPrice` / `getPremiumAmount`.
- OTM and range validations at mint: Checks that the option is OTM and within signed bounds compare `tickLower/tickUpper` with `slot0().tick` or compare `slot0().sqrtPriceX96` to signed min/max.
- Settlement/exercise math: In-range paths call helpers like `LiquidityAmounts.getAmountsForLiquidity(_getCurrentSqrtPriceX96(pool), ...)` where `_getCurrentSqrtPriceX96` reads `slot0()`.

Typical pattern:

```
(uint160 sqrtPriceX96, int24 tick,,,,,) = pool.slot0();  // instantaneous spot
// used to derive currentPrice, OTM checks, and settlement amounts
```

The Uniswap v3 spot can be moved within a single transaction via a large swap and then reverted, leaving no lasting cost to the attacker beyond fees and slippage. Using `slot0` directly lets an attacker:

- Underprice premiums: momentarily push spot down (for calls) or up (for puts) to mint options cheaply while still meeting OTM checks. After the block, spot normalizes and the option is mispriced in the attacker's favor. (Partially mitigated by the enforced signature).
- Bypass intended guards: OTM/range checks pegged to `slot0` can be satisfied during a transient spike/dip even though the TWAP market never truly reflected that state.
- Influence settlement amounts: any path that derives `amount0`/`amount1` from the current price can be gamed by nudging `slot0` at settlement time to optimize payouts or refunds.

While whitelisted signatures constrain some ranges, the on-chain checks remain based on the manipulable spot. On thinner pools or long-tail pairs this attack becomes inexpensive and repeatable. On deep pools it's still feasible around volatile periods.

Recommendation: Use Uniswap v3's built-in TWAP oracle instead of `slot0` for any price-sensitive logic. Replace spot reads with TWAP:

```
// Uniswap v3 periphery's OracleLibrary
(int24 meanTick, ) = OracleLibrary.consult(pool, SECONDS_AGO); // e.g., 300-900s
uint160 twapSqrtPriceX96 = TickMath.getSqrtRatioAtTick(meanTick);
// If you need a quote:
uint256 twapQuote = OracleLibrary.getQuoteAtTick(
    meanTick, baseAmount, baseToken, quoteToken
);
```

Choose `SECONDS_AGO` long enough to make single-block manipulation uneconomical (commonly 5-15 minutes for quotes/premiums).

Add deviation checks: `if abs(spot - twap)/twap > MAX_DEVIATION, revert`. This blocks transactions during manipulated blocks while still allowing healthy markets to function.

Harden all call sites:

- Premium/fee computation (getOptionPrice / getPremiumAmount) → TWAP.
- OTM / range enforcement at mint and exercise → compare against TWAP tick.
- Settlement amount math that depends on "current price" → compute using TWAP.

Ensure oracle capacity: on initialization, require `observationCardinalityNext` to be sufficiently high (e.g., ≥ 64) or call `increaseObservationCardinalityNext` so the pool can support your chosen TWAP window.

Stryke: Acknowledged. This attack is prevented by our firewall system, which checks for a price tolerance on top of the spot price via offchain data, the spot price oracle used here is one of the oracles that we use. The order depends on the best oracle available, Chainlink, TWAP, Spot. All of them have to be passed via the firewall, giving us additional protection on top of all the oracles we use, inclusive of the spot.

Cantina Managed: Acknowledged.

3.2.5 Last-element hook is used for the whole option, ignoring earlier entries

Severity: Medium Risk

Context: OptionMarketOTMFE.sol#L370

Description: In the `mintOption` function the contract reads `opTick.hook` from the last element of `_params.optionTicks` and uses it as the effective hook, silently discarding the hook values of all prior entries.

Currently, this may not be exploitable if all option ticks are expected to share the same hook and downstream logic does not branch on hook. However, the project already confirmed that hooks can determine pricing/IV. Under that design, the "last-element only" behavior becomes a pricing foot-gun:

- A user supplies many `optionTicks` that source liquidity under Hook A (higher IV / higher premium), and appends a tiny final tick that uses Hook B (lower IV).
- Because only the last element's hook is used, the premium can be computed with Hook B while most of the position's risk is under Hook A, producing systematic underpricing.
- Merely re-ordering `optionTicks` can change the premium and behavior, which should never happen.

Recommendation: Consider enforcing that all ticks share the same hook and use that hook explicitly.

Stryke: Acknowledged. We don't have any plans to use the hook for pricing, the current codebase also doesn't rely on hook pricing, tough we will also notify in the docs that, if you are using the hooks for pricing you need to preserve the order via an additional check on the option calldata or update us for the firewall for hook order preservation.

Cantina Managed: Acknowledged.

3.2.6 Strike tick validation bypass through incorrect distance calculation

Severity: Medium Risk

Context: OptionMarketOTMFE.sol#L293

Description: When validating the tick range width in `mintOption()`, the contract uses different logic depending on whether both ticks have the same sign or different signs:

```
if (opTick.tickUpper > 0 && opTick.tickLower > 0 || opTick.tickUpper < 0 && opTick.tickLower < 0) {
    if (uint24(opTick.tickUpper - opTick.tickLower) > maxTickDiff) {
        revert NotValidStrikeTick();
    }
} else {
    if (uint24(opTick.tickUpper + opTick.tickLower) > maxTickDiff) {  // Incorrect calculation
        revert NotValidStrikeTick();
    }
}
```

The issue occurs in the `else` block, which handles cases where ticks straddle zero (one negative, one positive). The code adds the tick values instead of calculating the actual distance between them.

For example:

- `tickLower = -1000`, `tickUpper = 1000`: Actual distance is 2,000, but code calculates 0.

This allows positions with tick ranges up to twice the intended `maxTickDiff` to pass validation when they straddle zero.

Recommendation: Simplify the logic to always use subtraction since `tickUpper` should always be greater than `tickLower`:

```
if (uint24(opTick.tickUpper - opTick.tickLower) > maxTickDiff) {
    revert NotValidStrikeTick();
}
```

Stryke: Fixed in commit 71173d9.

Cantina Managed: Fix verified.

3.2.7 Unsafe ERC20 operations can cause transaction failures with non-standard tokens

Severity: Medium Risk

Context: OptionMarketOTMFE.sol#L509, OptionMarketOTMFE.sol#L577, MintOptionFirewall.sol#L128, MintOptionFirewall.sol#L131, MultiSwapRouter.sol#L78, MultiSwapRouter.sol#L81, OnSwapReceiver.sol#L93

Description: The protocol directly calls ERC20 methods without using safe wrappers that handle non-standard token implementations. While the ERC20 standard specifies that `transfer`, `transferFrom`, and `approve` should return a boolean value, some major tokens like USDT don't follow this specification. The issue appears in multiple locations throughout the codebase, for example:

```
if (ac.totalProfit > 0) {
    ac.assetToGet.transfer(msg.sender, ac.totalProfit);
}
```

When Solidity code is compiled against the standard IERC20, the compiler will add logic to try to decode a boolean that isn't there, and the call will revert.

Recommendation: Use OpenZeppelin's SafeERC20 library throughout the codebase to handle both standard and non-standard token implementations safely. The SafeERC20 library provides wrapper functions that handle the return value checking gracefully.

Replace `transfer` and `transferFrom` with `safeTransfer` and `safeTransferFrom`. For approval operations, use `safeIncreaseAllowance`, `safeDecreaseAllowance`, or `forceApprove` depending on the specific use case.

Stryke: Fixed in commit 400bbc0.

Cantina Managed: Fix verified.

3.2.8 Premium/exercise mismatch within Uniswap V3 tick bands

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When minting options backed by a Uniswap V3 range, the premium for calls is computed using the upper tick as the strike price, while early exercise/settlement is only permitted once the pool's spot price falls below the lower tick. This creates a structural mismatch:

- Pricing side (calls):

  ```
  uint256 strike = getPricePerCallAssetViaTick(pool, tickUpper); // K from upper boundary
  ```

- Exercise side (calls): `ExerciseOptionFirewall.exerciseOption` will only work if the pool's spot price is below the option's lower tick.

The instrument sold is not a vanilla call struck at `upperTick` as it behaves like a down-and-in barrier call whose barrier is at `lowerTick`. A barrier reduces the chance the option can realize value before expiry, so the fair price is strictly lower than a vanilla call's premium. Charging a premium based on `upperTick` while gating early settlement at `lowerTick` systematically overcharges call buyers.

Because Uniswap V3 tick spacing is discrete, the magnitude of this overcharge is bounded if the protocol restricts the band width to at most one spacing step per pool:

Fee tier	Tick spacing	Price change across one step	Approx. strike bias if you use the upper instead of the band's mid
0.01%	1	~0.01%	~0.005%
0.05%	10	~0.10%	~0.05%
0.3%	60	~0.60%	~0.30%
1%	200	~2.02%	~1.01%

(*Each tick is a 0.01% multiplicative move; across Δ ticks, factor ≈ 1.0001^Δ. Using the band's upper boundary instead of the mid shifts the effective strike by roughly half the band width*)

In practical terms, on a 1 WETH notional at ~$3,500, a 200-tick band leads to ~$35 plus the barrier effect (the buyer can't exercise unless spot price is below the lower tick), which further depresses fair value. For 10-60 tick bands, the drift is smaller (5-30 bps), but still systematic.

Therefore traders pay too much premium, especially on wider bands or higher tick-spacing pools. This is not an acute "funds-at-risk" bug, but a persistent value transfer from buyers to writers that may not be obvious to users, degrading market fairness and UX. This issue is also present (in the opposite direction) for put options.

Recommendation: Consider pricing the premium of the call options at the `tickLower` and the put options at the `tickHigher` by updating the following line in the `mintOption` function:

```
- uint256 strike = getPricePerCallAssetViaTick(primePool, _params.isCall ? _params.tickUpper :
↪  _params.tickLower);
+ uint256 strike = getPricePerCallAssetViaTick(primePool, _params.isCall ? _params.tickLower :
↪  _params.tickUpper);
```

Stryke: Fixed in commit 7213104.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 `_getPrice` can silently return zero at extreme ticks due to integer flooring

Severity: Low Risk

Context: OptionMarketOTMFE.sol#L693-L710, PoolSpotPrice.sol#L19-L37

Description: The helper that derives a quoted price from the Uniswap V3 sqrt price uses integer arithmetic that floors toward zero. For certain (extreme) `sqrtPriceX96` values the computation underflows to 0, even though the pool state is valid.

The exact thresholds depend on decimals and whether `callAsset` equals `token0` or `token1`. For example:

For `callAsset == token0`.

- With 6 decimals: returns 0 for $S \leq 792281625142643375935439 50$.
- With 18 decimals: returns 0 for $S \leq 79228162514264337593$.

For `callAsset == token1`.

- With 6 decimals: returns 0 for $S \geq 792281625142643375935439 50336001$.
- With 18 decimals: returns 0 for $S \geq 792281625142643375935439 50336000000001$.

These values are far from typical market conditions, but the failure mode is silent: downstream code that assumes a positive price can mis-quote premiums (e.g., zero/near-zero premium), mis-evaluate OTM checks, or later hit divisions by zero when a non-zero price was expected. The risk also increases for long-tail tokens with unusual decimal configurations.

Recommendation: There is no straightforward fix to this issue. The best option would be enforcing safe tick bounds at mint/settle by rejecting positions when `_getPrice` return a 0 price.

Stryke: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 EIP-712 RangeCheck typehash/struct mismatch breaks signature verification

Severity: Low Risk

Context: ExerciseOptionFirewall.sol#L55-L57, MintOptionFirewall.sol#L59-L61

Description: In the firewall contracts the declared EIP-712 type string for `RangeCheck` does not match what is actually encoded in `_checkRange`. The constant is missing the market field and has a misspelled field name (`maxSprtPriceX96`), while the code that builds the struct hash encodes an extra market field and uses the correct spelling `maxSqrtPriceX96`. As a result, off-chain signers that follow the declared type string will compute a different digest than the contract expects, causing signature verification to fail.

Declared typehash:

```
bytes32 private constant RANGE_CHECK_TYPEHASH = keccak256(
  "RangeCheck(address user,address pool,int24 minTickLower,int24 maxTickUpper,uint160 minSqrtPriceX96,uint160
  ↪  maxSprtPriceX96,uint256 deadline)"
);
```

On-chain encoding:

```
// Pseudocode of _checkRange packing
abi.encode(
  RANGE_CHECK_TYPEHASH,
  user,
  pool,
  market,              // <- extra field encoded
  minTickLower,
  maxTickUpper,
  minSqrtPriceX96,
  maxSqrtPriceX96,     // <- corrected spelling expected here
  deadline
);
```

This mismatch can create a functional DoS for any flow gated by this signature, because valid, standards-compliant EIP-712 signatures will consistently be rejected.

This inconsistency exists in both `ExerciseOptionFirewall` and `MintOptionFirewall`.

Recommendation: Make the type string match the encoded fields exactly and standardize across both firewalls.

```
bytes32 private constant RANGE_CHECK_TYPEHASH = keccak256(
  "RangeCheck(address user,address pool,address market,int24 minTickLower,int24 maxTickUpper,uint160
  ↪  minSqrtPriceX96,uint160 maxSqrtPriceX96,uint256 deadline)"
);
```

And ensure the struct hash uses the same ordering:

```
bytes32 structHash = keccak256(abi.encode(
  RANGE_CHECK_TYPEHASH,
  rc.user,
  rc.pool,
  rc.market,
  rc.minTickLower,
  rc.maxTickUpper,
  rc.minSqrtPriceX96,
  rc.maxSqrtPriceX96,
  rc.deadline
));
```

Stryke: Fixed in commit 612c90b.

Cantina Managed: Fix verified.

3.3.3 Incorrect argument passed to `LogWithdrawReserveLiquidity` event

Severity: Low Risk

Context: V3BaseHandler.sol#L533

Description: The `LogWithdrawReserveLiquidity` expects `amount0` and `amount1` but `amount1` is being passed twice.

Recommendation:

```
- emit LogWithdrawReserveLiquidity(_params, context, amount1, amount1);
+ emit LogWithdrawReserveLiquidity(_params, context, amount0, amount1);
```

Stryke: Fixed in commit 90e39ee.

Cantina Managed: Fix verified.

3.3.4 Unsafe downcast from in liquidity calculations

Severity: Low Risk

Context: OptionMarketOTMFE.sol#L466, OptionMarketOTMFE.sol#L493, OptionMarketOTMFE.sol#L498, OptionMarketOTMFE.sol#L519, OptionMarketOTMFE.sol#L524, OptionMarketOTMFE.sol#L545, OptionMarketOTMFE.sol#L558, V3BaseHandler.sol#L414, V3BaseHandler.sol#L550, V3BaseHandlerVe33Shadow.sol#L420, V3BaseHandlerVe33Shadow.sol#L559

Description: The contract performs unsafe type casting when converting liquidity values from uint256 to uint128:

```
// src/apps/options/OptionMarketOTMFE.sol:466
uint128(liquidityToSettle)
```

Solidity does not protect against overflow during type casting operations. When a uint256 value exceeds the maximum value of uint128 (2^128 - 1), the cast will silently overflow, resulting in an incorrect smaller value. This is particularly concerning for liquidity amounts, which can be substantially larger than individual token balances, especially for ultra-deep out-of-the-money options or in pools with significant capital concentration.

Recommendation: Consider implementing OpenZeppelin's SafeCast library to ensure safe type conversions throughout the codebase. This provides automatic overflow checking without significant gas overhead.

The SafeCast library (OpenZeppelin SafeCast) will revert with a clear error message if the value exceeds the target type's range, preventing silent overflow and making the protocol more robust against edge cases.

Stryke: Acknowledged. As we use only UniswapV3, the underlying UniswapV3 ensures for this. However, we will document this, incase there is some other dex used which doesn't not ensure this.

Cantina Managed: Acknowledged.

3.3.5 Incorrect tick boundary validation allows option minting at exact boundary

Severity: Low Risk

Context: OptionMarketOTMFE.sol#L300-L306, OptionMarketOTMFE.sol#L473

Description: The `OptionMarketOTMFE` contract contains validation logic to prevent minting options when the current pool price falls within the specified tick range. However, the validation uses strict less-than comparisons (<) for the lower bound, which incorrectly allows option minting when the pool price sits exactly on the `tickLower` boundary.

The problematic code checks:

```
opTick.tickLower < TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))
    && opTick.tickUpper > TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))
```

Since `getTickAtSqrtRatio()` floors to the nearest tick, when the pool price equals the exact price at `tickLower`, the current tick equals `tickLower`. The strict inequality fails to catch this boundary case, allowing options to be minted when they shouldn't be.

For out-of-the-money (OTM) options, the price should be strictly outside the tick range to maintain the option's OTM characteristic. Allowing minting at the exact boundary violates this constraint.

Recommendation: Update the boundary checks to use less-than-or-equal-to (<=) for the lower bound comparisons to properly reject options when the price sits exactly at the tick boundary:

```
  if (
      (
-         opTick.tickLower < TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))
+         opTick.tickLower <= TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))
              && opTick.tickUpper > TickMath.getTickAtSqrtRatio(_getCurrentSqrtPriceX96(opTick.pool))
      )
          || (
-             TickMath.getSqrtRatioAtTick(opTick.tickLower) < _getCurrentSqrtPriceX96(opTick.pool)
+             TickMath.getSqrtRatioAtTick(opTick.tickLower) <= _getCurrentSqrtPriceX96(opTick.pool)
                  && TickMath.getSqrtRatioAtTick(opTick.tickUpper) > _getCurrentSqrtPriceX96(opTick.pool)
          )
  ) {
      revert NotValidStrikeTick();
  }
```

Stryke: Fixed in commit 9daaacd.

Cantina Managed: Fix verified.

3.3.6 No per-account pro-rata withdrawal cap lets fast LPs drain the ”free bucket”

Severity: Low Risk

Context: V3BaseHandler.sol#L315-L319

Description: In `V3BaseHandler`, liquidity for a given (`pool`, `hook`, `tickLower`, `tickUpper`) is aggregated into a single `tokenId`. Applications (e.g., `OptionMarketOTMFE`) consume liquidity from this shared bucket by increasing `liquidityUsed`. When LPs withdraw, the handler only checks aggregate availability:

```
// src/handlers/V3BaseHandler.sol:315
TokenIdInfo storage tki = tokenIds[cache.tokenId];

if ((tki.totalLiquidity - tki.liquidityUsed) < cache.liquidity) {
    revert InsufficientLiquidity();
}
```

There is no per-account cap tied to a user's share of the position. As a consequence, whenever some liquidity is returned to the pool (e.g., an option settles and `liquidityUsed` decreases), the first LP to submit a withdrawal can take all of the newly freed liquidity, even if that exceeds their fair (pro-rata) share. Other LPs, who hold the same `tokenId` but are slower, are temporarily stranded until more liquidity is freed later.

This creates a race any time liquidity oscillates between ”used” and ”free”.

Recommendation: Consider implementing a per-account pro-rata withdrawal cap for each `tokenId`, so no LP can withdraw more than their fair share of the currently free liquidity.

Stryke: Acknowledged. This is an expected behaviour. The reserve liquidity is only used if the LP wants to exit before the intended expiry, and if so, they are penalized for exiting early as they lose all the premiums when they reserve the liquidity.

Cantina Managed: Acknowledged.

3.4 Informational

3.4.1 Missing event emission on critical parameter updates

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Several admin "setter" functions across the codebase change parameters that directly affect user pricing, permissions and payouts, but do not emit any event. Without events, indexers, UIs, risk monitors and users cannot reliably detect these changes. This weakens transparency around governance actions that change option premiums, decide who can sign or execute guarded actions or alter settlement fees and fee recipients.

Affected functions:

- `OptionPricingLinearV2`: `updateIVSetter`, `updateIVs`, `updateVolatilityOffset`, `updateVolatilityMultiplier`, `updateMinOptionPricePercentage`.
- `MintOptionFirewall`: `updateWhitelistedSigner`.
- `ExerciseOptionFirewall`: `updateWhitelistedExecutor`, `updateWhitelistedMarket`.
- `OpenSettlement`: `setSettleFeeProtocol`, `setSettleFeePublic`, `setIsWhitelistedSettler`, `setPublicFeeRecipient`.
- `V3BaseHandler`: `registerHook`, `updateHandlerSettings`, `emergencyPause`, `emergencyUnpause`.
- `BoundedTTLHook_0Day` and `BoundedTTLHook_1Week`: `updateWhitelistedAppsStatus`.
- `OptionMarketOTMFE.updatePoolSettings(...)` emits `LogUpdatePoolSettings(feeTo, tokenURIFetcher, dpFee, optionPricing)` but also updates `verifiedSpotPrice`, `maxTickDiff`, `maxUpperTick`, `minLowerTick` and `minLiquidityToUse` without including them in the event payload. Consider expanding the event so all changed fields are visible.

Recommendation: Emit explicit, indexed events for every admin change that alters protocol behavior or economics. Include the key/subject, the old and new values and the caller. For batch updates, emit one event per element so indexers can correlate precisely.

Stryke: Acknowledged.

Cantina Managed: Acknowledged.

3.4.2 View functions can be declared as pure

Severity: Informational

Context: OptionMarketOTMFE.sol#L226-L236

Description: In the `OptionMarketOTMFE` contract, the `name()` and `symbol()` functions are overridden from the ERC721 standard and declared as `view` functions. These functions return constant string values ("MarginZero Option Market OTM FE" for the `name` and "MZ-OM-OTM-FE" for the `symbol`) without reading any on-chain state or storage variables. As a result, they can be safely marked as `pure` instead of `view`, which would allow the Solidity compiler to optimize them further by enabling more aggressive constant folding and potentially reducing gas costs for external calls.

Recommendation: Update the function modifiers from `view` to `pure` to allow better compiler optimizations.

Stryke: Fixed in commit f124e20.

Cantina Managed: Fix verified.

3.4.3 Remove vestigial references to "delegate"

Severity: Informational

Context: OptionMarketOTMFE.sol#L83, OptionMarketOTMFE.sol#L108, OptionMarketOTMFE.sol#L589

Description: The code still contains vestigial references to delegate in an error and an unused event even though the concept is no longer used.

Recommendation: Rename `NotOwnerOrDelegator` to `NotOwner` and remove `LogUpdateExerciseDelegate` event.

Stryke: Fixed in commit f124e20.

Cantina Managed: Fix verified.

3.4.4 Remove unused event

Severity: Informational

Context: V3BaseHandler.sol#L135, OnSwapReceiver.sol#L41

Description: The code contains an unused, vestigial event `SwapperWhitelisted`.

Recommendation: Remove event.

Stryke: Fixed in commit f124e20.

Cantina Managed: Fix verified.

3.4.5 Incorrect comment

Severity: Informational

Context: BlackScholes.sol#L28, BlackScholes.sol#L45

Description: The comment says `days`, but the logic uses centidays (1/100 of day).

Recommendation: Correct the comment.

Stryke: Acknowledged.

Cantina Managed: Acknowledged.

3.4.6 Prefer custom error to raw revert

Severity: Informational

Context: OptionPricingLinearV2.sol#L257

Description: A raw revert is used here which does not provide any information and makes debugging more difficult.

Recommendation: Consider adding a custom error.

Stryke: Acknowledged.

Cantina Managed: Acknowledged.

3.4.7 Lack of a double-step transfer ownership pattern

Severity: Informational

Context: OptionMarketOTMFE.sol#L29, OptionPricingLinearV2.sol#L11, BoundedTTLHook_0Day.sol#L12, BoundedTTLHook_1Week.sol#L12, ExerciseOptionFirewall.sol#L15, MintOptionFirewall.sol#L17, OpenSettlement.sol#L13, PositionManager.sol#L18, OnSwapReceiver.sol#L14

Description: The Stryke protocol uses the `Ownable` library in multiple contracts. The standard OpenZeppelin's Ownable contract allows transferring the ownership of the contract in a single step:

```
/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}
```

```
/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

Recommendation: It is recommended to implement a two-step transfer process in all the contracts in the codebase where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. A good code example could be OpenZeppelin's Ownable2Step contract:

```
/**
 * @dev Starts the ownership transfer of the contract to a new account. Replaces the pending transfer if there is
 ↪  one.
 * Can only be called by the current owner.
 *
 * Setting `newOwner` to the zero address is allowed; this can be used to cancel an initiated ownership transfer.
 */
function transferOwnership(address newOwner) public virtual override onlyOwner {
    _pendingOwner = newOwner;
    emit OwnershipTransferStarted(owner(), newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`) and deletes any pending owner.
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual override {
    delete _pendingOwner;
    super._transferOwnership(newOwner);
}

/**
 * @dev The new owner accepts the ownership transfer.
 */
function acceptOwnership() public virtual {
    address sender = _msgSender();
    if (pendingOwner() != sender) {
        revert OwnableUnauthorizedAccount(sender);
    }
    _transferOwnership(sender);
}
```

Stryke: Acknowledged. We will be careful when transferring the ownership.

Cantina Managed: Acknowledged.

3.4.8 `ExerciseOptionFirewall.settleOption` reverts in-range due to missing allowances

Severity: Informational

Context: ExerciseOptionFirewall.sol#L109-L131

Description: `ExerciseOptionFirewall.settleOption` is a privileged entrypoint gated by `whitelistedExecutors`, but it does not enforce any price-range or expiry precondition before delegating to the core market. In practice, when the option is in-range pre-expiry, the market's `settleOption` path requires the caller to fund part of the settlement (e.g., it safeTransferFrom(msg.sender, ...) the non-call asset during in-range exercise/settlement). Because the firewall contract does not maintain balances nor set allowances to the market, this path reverts.

Recommendation: Consider documenting this behaviour in your contracts and in the protocol documentation. Another option is adding tick range checks by requesting a valid signature and let the LPers call it after expiry.

Stryke: Acknowledged. This is expected, as we do not want this contract to be able to settle any in-range position.

Cantina Managed: Acknowledged.

3.4.9 Misleading option-type flag comment contradicts BlackScholes interface

Severity: Informational

Context: OptionPricingLinearV2.sol#L187

Description: In `OptionPricingLinearV2` the inline comment next to the option-type argument is inverted relative to the actual `BlackScholes.calculate` interface:

```
// OptionPricingLinearV2.sol:187 (approx.)
uint256 optionPrice = BlackScholes.calculate(
    _params.isPut ? 1 : 0, // 0 - Put, 1 - Call  <-- incorrect comment
    _params.lastPrice,
    _params.strike,
    timeToExpiry,
    0,
    volatility
);
```

Per the `BlackScholes` library used in this code path, the flag mapping is `0 = Call`, `1 = Put`. The implementation correctly passes `_params.isPut ? 1 : 0`, but the comment states the opposite ("`0 - Put, 1 - Call`"), which is misleading.

Impact for users / protocol risk.

Recommendation: Correct the comment to reflect the library's contract:

```
// optionType: 0 = Call, 1 = Put
```

Stryke: Fixed in commit 8543f5d.

Cantina Managed: Fix verified.

3.4.10 Incorrect comment in `minOptionPricePercentage`

Severity: Informational

Context: OptionPricingLinearV2.sol#L20

Description: `OptionPricingLinearV2` documents `minOptionPricePercentage` as "in 1e8 precision", but the implementation divides by 1e10 when computing the minimum option price:

```
// OptionPricingLinearV2.sol:20
// The % of the price of asset which is the minimum option price possible in 1e8 precision

// ... later ...
uint256 minOptionPrice = _params.lastPrice
    .mul(minOptionPricePercentage[_params.optionsMarket])
    .div(1e10); // ← implementation uses 1e10 scale
```

Operationally, this field is a configuration knob used by governance/ops to enforce a floor on option prices. Because the comment (and likely any derived docs/internals) states 1e8 but the arithmetic expects 1e10, an operator setting values under the 1e8 assumption will unintentionally program a value 100 times smaller than intended.

Recommendation: Update the comment to:

```
// OptionPricingLinearV2.sol:20
// The % of the price of asset which is the minimum option price possible in 1e10 precision
```

Stryke: Fixed in commit 8543f5d.

Cantina Managed: Fix verified.

3.4.11 Incorrect use of shadow variable

Severity: Informational

Context: LiquidityManager.sol#L121

Description: The function signature accepts `_factory` but the body ignores it and uses the state variable `factory`. The signature implies you can compute/verify against any factory, but the implementation forces the use of the contract's `factory` immutable variable. It works for current usage because `factory` is always passed as `_factory` in `verifyCallback`, but it's an inconsistency and could break future reuse. It's also inconsistent with the Uniswap/Sushi variants which do use the correct parameter.

Recommendation: Correct the variable.

```
- factory,
+ _factory,
```

Stryke: Acknowledged.

Cantina Managed: Acknowledged.

3.4.12 MintOptionFirewall over-collects maxCostAllowance and doesn't refund the unused amount

Severity: Informational

Context: MintOptionFirewall.sol#L126-L132

Description: `MintOptionFirewall.mintOption` transfers the entire `maxCostAllowance` from the user to the firewall and approves the market for that amount. During the `OptionMarketOTMFE.mintOption` call, the protocol computes the actual premium and protocol fee, then pulls only those amounts from the firewall via `transferFrom`. The remainder of the user's funds stays stranded on the firewall and is not refunded.

This behavior is visible in the provided POC-trace for a PUT mint. The firewall first receives 48.703357 USDC from the user (`transferFrom(user $\rightarrow$ MintOptionFirewall, 48703357)`). Inside `OptionMarketOTMFE.mintOption`, the premium is calculated at 11.598210 USDC and the protocol fee at 4.059373 USDC. The market pulls exactly those two amounts (`transferFrom(MintOptionFirewall $\rightarrow$ OptionMarketOTMFE, 11598210)` and `transferFrom(MintOptionFirewall $\rightarrow$ feeReceiver, 4059373)`), totaling 15.657583 USDC. The leftover 33.045774 USDC remains in the firewall and is never returned to the user.

This USDC stuck in the `MintOptionFirewall` can be used by any user that backruns the previous call to purchase options himself through the `self=true` flow.

Because of this, users can be systematically overcharged and lose the difference between `maxCostAllowance` and the actual `premium+fees` on every mint. Funds accumulated on the firewall can be used by malicious users to mint options for free.

Proof of Concept: See gist 7106d009.

The test mints a PUT option via `MintOptionFirewall.mintOption`. It sets `maxCostAllowance` to the user's "expected cost" (`premium + fee`) as computed off-chain/externally. The PoC then traces token flows during `mintOption` to show that the firewall pre-pulls the full allowance, while the market only consumes the actual `premium + protocol fee`, leaving the unused remainder stuck in the firewall.

Flow:

1. User funding & approval: The user funds USDC and approves the firewall for `maxCostAllowance` (e.g., 48.703357 USDC in your run).
2. Firewall pre-pulls full allowance: `MintOptionFirewall.mintOption` calls `USDC.transferFrom(user $\rightarrow$ firewall, maxCostAllowance)`.
3. Market calculates actual costs: Inside `OptionMarketOTMFE.mintOption`, the premium is computed by `OptionPricingLinearV2.getOptionPrice(...)` and the protocol fee by `ClammFeeStrategyV2.onFeeReqReceive(...)`. In this test: premium ≈ 11.598210 USDC, protocol fee ≈ 4.059373 USDC.
4. Market pulls only what it needs from the firewall. The market then pulls these amounts from the firewall:

- `transferFrom(MintOptionFirewall $\rightarrow$ OptionMarketOTMFE, 11.598210)` (premium).

- `transferFrom(MintOptionFirewall $\rightarrow$ feeReceiver, 4.059373)` (protocol fee).
- Total consumed ≈ 15.657583 USDC.

5. The remainder is never refunded: The difference (48.703357 − 15.657583 = 33.045774 USDC) remains at the `MintOptionFirewall` address. There is no refund logic in `MintOptionFirewall.mintOption` and no subsequent transfer returning the balance to the user. The option NFT is minted and transferred, but the excess user funds stay stranded on the firewall.

Recommendation: Do not pre-pull the full maxCostAllowance without reconciliation. Either quote costs first and pull exactly the required amount, or if pre-pulling is retained, refund the residual immediately after mint based on the firewall's token balance delta.

Stryke: Acknowledged. This is not a problem for us as this is mentioned in the docs where there is a comment about using the sweep function (MintOptionFirewall.sol#L157) to be batched after every swap or `mintOption` operation, in order to transfer any residue tokens back to the user.

Cantina Managed: Acknowledged.

3.4.13 Missing validation in `PoolSpotPrice._getPrice` function

Severity: Informational

Context: PoolSpotPrice.sol#L19

Description: In the `PoolSpotPrice` contract, the function `_getPrice` accepts a `callAsset` address but never verifies that this asset is one of the pool's tokens. If a wrong address is passed, the function will still compute and return a price without reverting. Because price direction and scaling depend on the pool's token0/token1 ordering and their decimals, using an asset that is not equal to `_pool.token0()` nor `_pool.token1()` yields a price in the wrong units and/or for the wrong base/quote relationship. The mistake is silent and propagates into premium, strike and fee calculations (e.g., via `OptionMarketOTMFE` and pricing modules), resulting in systematic over- or under-charging.

Recommendation: Add a strict membership check:

```
error InvalidCallAsset(address callAsset, address token0, address token1);

require(
    callAsset == _pool.token0() || callAsset == _pool.token1(),
    InvalidCallAsset(callAsset, _pool.token0(), _pool.token1())
);
```

Stryke: Acknowledged. A wrong address will never be passed as it is only used from the option market.

Cantina Managed: Acknowledged.