Clove

Security Review

Cantina Managed review by:
R0bert, Lead Security Researcher
Jonatas Martins, Security Researcher

February 24, 2026

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 level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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 are 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

From Feb 15th to Feb 18th the Cantina team conducted a review of clob on commit hash 7256e3df. The team identified a total of **9** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	0	0	0
Low Risk	3	3	0
Gas Optimizations	0	0	0
Informational	4	4	0
Total	**9**	**9**	**0**

2.1 Scope

The security review had the following components in scope for clob on commit hash 7256e3df:

```
contracts/src/v2
├── AccountLedger.sol
├── NonceManager.sol
├── SpotSettlement.sol
├── VaultV2.sol
└── interfaces
    ├── IAccountLedger.sol
    ├── IFundingModule.sol
    ├── ILiquidationModule.sol
    ├── IPerpClearing.sol
    └── IVaultV2.sol
```

3 Findings

3.1 High Risk

3.1.1 Canonical fill IDs and nonce checks missing

Severity: High Risk

Context: AccountLedger.sol#L239-L265, NonceManager.sol#L35-L50, SpotSettlement.sol#L85-L86, SpotSettlement.sol#L161-L168, SpotSettlement.sol#L170-L176, SpotSettlement.sol#L186-L203

Description: `SpotSettlement.settleSpotBatch` tracks replay with `fillUsage[fillId]` and writes the mapping entry directly from the `fillId` value supplied in each `MatchedFill`. The function does not derive a canonical fill identifier from the full match semantics before checking replay state, which allows equivalent settlement semantics to be replayed with fresh `fillId` values while keeping the operator-level checks valid.

```
for (uint256 i = 0; i < fills.length; i++) {
    bytes32 fillId = fills[i].fillId;
    uint256 priorBatch = fillUsage[fillId];
    if (priorBatch != 0) revert FillAlreadyConsumed(fillId, priorBatch);
    fillUsage[fillId] = batch.batchId;
    emit FillConsumed(fillId, batch.batchId);
}
```

The same function does not consume `NonceManager` nonces or verify order intent expiry for either side of a match, so replay prevention is disconnected from user-intent state.

```
function verifyOrderIntent(OrderIntent calldata intent, bytes calldata sig) external
↪  view returns (bool) {
    bytes32 structHash = keccak256(
        abi.encode(
            ORDER_INTENT_TYPEHASH,
            intent.user,
            intent.baseToken,
            intent.quoteToken,
            intent.amount,
            intent.limitPrice,
            intent.nonce,
            intent.expiry,
            intent.isBuy
        )
    );
    bytes32 digest = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR,
    ↪  structHash));
    return _recoverSigner(digest, sig) == intent.user;
}
```

```
function useNonce(address user, uint256 nonce) external {
    uint256 floor = minValidNonce[user];
    if (nonce < floor) revert NonceTooLow(floor, nonce);
    if (nonceUsed[user][nonce]) revert NonceAlreadyUsed(user, nonce);
}
```

This is critical because valid economic matches can be represented under alternate identifiers and still advance state when user-level replay protections are not enforced in settlement execution.

There is a second gap in the same settlement path. After the replay checks, `SpotSettlement` forwards `batch.deltas` directly to `AccountLedger` without any on-chain check that the deltas are economically consistent with the provided fills and without any per-token conservation constraint. That means replay protection alone does not prevent an authorized operator from submitting internally valid signatures over economically invalid balance transitions.

```
IAccountLedger.Delta[] memory ledgerDeltas = new
↪  IAccountLedger.Delta[](batch.deltas.length);
//...
```

```
IAccountLedger(accountLedger).applyNetDeltas(ledgerDeltas, batch.batchId);
```

Recommendation: Derive a canonical fill hash inside settlement and use that hash as the replay key. Add intent signature verification and expiry checks for both participants, and consume nonces through `NonceManager` before marking the canonical fill as used.

In the same hardening pass, add on-chain consistency checks that bind deltas to validated fill economics and enforce conservation constraints before calling `AccountLedger.applyNetDeltas`.

```
bytes32 canonicalFillId = keccak256(
    abi.encode(
        fills[i].maker,
        fills[i].taker,
        fills[i].makerOrderHash,
        fills[i].takerOrderHash,
        fills[i].baseAmount,
        fills[i].quoteAmount
    )
);
if (fillUsage[canonicalFillId] != 0) revert FillAlreadyConsumed(canonicalFillId,
↪  fillUsage[canonicalFillId]);
nonceManager.useNonce(maker.user, makerIntent.nonce);
fillUsage[canonicalFillId] = batch.batchId;
```

Clove: Fixed in PR 12.

Cantina Managed: Fix verified. Canonical fill replay and token conservation were implemented, but nonce integration into settlement execution was not wired in, so the issue is not 100% addressed.

3.1.2 `SetupProd` deploys publicly mintable collateral

Severity: High Risk

Context: SetupProd.s.sol#L51-L54

Description: The production deployment script instantiates `MockERC20` contracts as core collateral assets. `MockERC20` exposes an unrestricted `mint` function, so any caller can create arbitrary balances. In an isolated local environment this can be considered intentiona but in a production-labeled workflow this is a critical trust failure because collateral integrity is no longer anchored to a real asset supply.

```
MockERC20 usdc = new MockERC20("USD Coin", "USDC", 6);
MockERC20 weth = new MockERC20("Wrapped Ether", "wETH", 18);
MockERC20 wbtc = new MockERC20("Wrapped Bitcoin", "wBTC", 8);
```

```
function mint(address to, uint256 amount) external {
    _mint(to, amount);
}
```

If this script is ever used on a shared testnet or live environment, any external actor can mint collateral-like tokens, deposit them and drain honest reserves through normal withdrawal paths.

Impact is high because it can invalidate the protocol's entire accounting model and compromise all counterparties relying on those balances.

Recommendation: Ensure these `MockERC20` contracts are not used for production deployments and deploy `ERC20` contracts instead.

Clove: Fixed in PR 13 (commit 28027e40). `SetupProd.s.sol` is renamed to `SetupDev.s.sol` and now contains explicit warnings about unrestricted mock minting.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 `ReentrancyGuard` on Storage-Only Functions

Severity: Low Risk

Context: AccountLedger.sol#L195-L265

Description: The functions `creditDeposit`, `lock`, `unlock` and `applyNetDeltas` currently use `nonReentrant` even though they perform only internal state updates and do not invoke external callbacks. Entering and exiting `ReentrancyGuard` writes to and reads dedicated storage, which adds fixed overhead on every call even when no reentrant surface exists. In high-volume settlement and balance update flows this overhead compounds and adds non-trivial gas cost without increasing safety because withdrawals are already protected by separate checks and external-call paths.

Removing the modifier from these storage-only functions preserves behavior and lowers gas consumption on every call path.

Recommendation: Restrict `nonReentrant` to paths that perform external calls to `vault` and keep a minimal set of guarded entry points. For example:

```
function creditDeposit(address user, address token, uint256 amount)
    external
    onlyRole(VAULT_ROLE)
    whenNotPaused
{
//...
}
```

Clove: Fixed in PR 14 (commit a69aa3fa). `AccountLedger.creditDeposit`, `lock`, `unlock` and `applyNetDeltas` no longer use `nonReentrant` in PR 14, matching the storage-only optimization request.

Cantina Managed: Fix verified. The same PR removes unnecessary `nonReentrant` from the storage-only functions while preserving it on the external-call withdrawal paths (`debitForWithdrawal` / `debitForWithdrawalWithUnwrap`).

3.2.2 Custom ECDSA recovery without malleability checks

Severity: Low Risk

Context: SpotSettlement.sol#L268-L286

Description: `SpotSettlement._recoverSigner(...)` is a custom signer recovery helper and is also used indirectly by `verifyOrderIntent`. The implementation accepts a 65-byte signature after basic parsing and passes it to `ecrecover` without low-`s` enforcement.

```
function _recoverSigner(bytes32 hash, bytes calldata signature) internal pure returns
↪   (address) {
    if (signature.length != 65) revert InvalidOperatorSignature();

    bytes32 r;
    bytes32 s;
    uint8 v;

    assembly {
        r := calldataload(signature.offset)
        s := calldataload(add(signature.offset, 32))
        v := byte(0, calldataload(add(signature.offset, 64)))
    }

    if (v < 27) v += 27;

    address recovered = ecrecover(hash, v, r, s);
    if (recovered == address(0)) revert InvalidOperatorSignature();
    return recovered;
}
```

Signature malleability through high-`s` variants is not prevented. In many current flows this may be harmless, but it becomes risky if signatures are treated as unique identifiers or persisted as proof artifacts.

Recommendation: Consider using `ECDSA.recover` from OpenZeppelin for standardized recovery semantics. This enforces canonical signatures, including valid `v` handling and low-`s` checks.

```
using ECDSA for bytes32;
address recovered = hash.toEthSignedMessageHash().recover(signature);
```

If your stack uses short signatures, add explicit support for EIP-2098 packed signatures alongside the 65-byte path.

Clove: Fixed in PR 14 (commit a69aa3fa). `SpotSettlement._recoverSigner` now uses `ECDSA.tryRecover`, replacing the custom parser/recover logic with OpenZeppelin standard recovery checks (including low-`s`/`v` validation).

Cantina Managed: Fix verified. The custom hand-rolled recovery path was replaced by OZ `ECDSA` recovery in PR 14.

3.2.3 Immutable `DOMAIN_SEPARATOR` can allow signature replay across hard-forks

Severity: Low Risk

Context: SpotSettlement.sol#L38

Description: The `DOMAIN_SEPARATOR` is set as immutable, if a hard fork results in a chain-id change, the contract may continue validating signatures tied to the pre-fork domain, which enables replay attack across forked chains.

Recommendation: Derive the `DOMAIN_SEPARATOR` from the current block.chainid (or cache it with the chain id and recompute when block.chainid changes), following the `EIP-712` pattern used by Openzeppelin.

Clove: Fixed in PR 21.

Cantina Managed: Fix verified. The domain separator replay exposure across chain-id changes was addressed with a fork-safe/chain-id-aware EIP-712 pattern.

3.3 Informational

3.3.1 Zero address can brick Exchange roles

Severity: Informational

Context: SpotSettlement.sol#L110-L112

Description: `SpotSettlement` grants role state in constructor from the `admin` and `operator` parameters without zero-address validation. If any are set to zero, role recovery and emergency controls can become impossible depending on which constructor argument is affected.

```
constructor(address admin, address operator, address initialAccountLedger, address
↪  initialNonceManager) {
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(PAUSER_ROLE, admin);
    _grantRole(OPERATOR_ROLE, operator);

    accountLedger = initialAccountLedger;
    nonceManager = initialNonceManager;
}
```

Because there is no zero guard, governance and operator pathways can be permanently unusable from an invalid deployment configuration.

Recommendation: Fail deployment on zero-address role parameters for both `admin` and `operator`, and apply the same pattern to any other security-critical role-initializing constructor.

```
if (admin == address(0) || operator == address(0)) revert ZeroAddress();
```

Clove: Fixed in PR 15. `SpotSettlement` now guards admin and operator constructor args with `ZeroAddress`, so role assignment can't be initialized to `address(0)` and permanently brick role controls.

Cantina Managed: Fix verified. This matches the issue's exact remediation (`admin == 0` or `operator == 0` now reverts before role grants), so the zero-address role-bricking condition is addressed in that PR.

3.3.2 Zero address can brick Vault roles

Severity: Informational

Context: VaultV2.sol#L54-L55

Description: `VaultV2` constructor assigns `DEFAULT_ADMIN_ROLE` and `PAUSER_ROLE` directly from the `admin` argument and `initialLedger` is optional only by zero check. A zero admin leaves role administration and policy control unassignable.

```
constructor(address admin, address initialLedger) {
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(PAUSER_ROLE, admin);
    if (initialLedger != address(0)) {
        accountLedger = initialLedger;
    }
}
```

A misconfigured zero role address can make role recovery and operational recovery actions unreachable for the vault path.

Recommendation: Consider adding the following checks in the constructor:

```
if (admin == address(0)) revert ZeroAddress();
if (initialLedger != address(0) && initialLedger.code.length == 0) revert
↪  InvalidAddress();
```

Clove: Fixed in PR 15. `VaultV2` now reverts if `admin == address(0)` in the constructor, so role setup for `DEFAULT_ADMIN_ROLE` / `PAUSER_ROLE` can't be initialized to zero and brick role control.

Cantina Managed: Fix verified.

3.3.3 Vault deposit assumes full amount received

Severity: Informational

Context: VaultV2.sol#L63-L77

Description: `VaultV2.deposit` credits internal accounting with the user-declared `amount` and then performs `safeTransferFrom` with that same amount, without measuring the effective token delta on the vault address. If a token has transfer tax, rebasing behavior or non-standard accounting, the vault can receive less than `amount` while internal balances still record a full credit.

Recommendation: If you want to support fee on transfer tokens consider computing the actual received values and use `balanceAfter - balanceBefore` as the credited amount, or restrict the supported token set to standard fixed-supply transfer semantics.

```
uint256 before = IERC20(token).balanceOf(address(this));
IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
uint256 received = IERC20(token).balanceOf(address(this)) - before;
totalTokenBalance[token] += received;
IAccountLedger(accountLedger).creditDeposit(receiver, token, received);
```

Clove: Fixed in PR 22 (commit c1c12dec). `VaultV2.deposit` now computes `balanceOf` before/after transfer and credits only the actual received amount; this directly addresses transfer-fee/non-standard token behavior.

Cantina Managed: Fix verified. The PR includes `VaultV2` delta-based deposit accounting and adds tests for fee-on-transfer handling (`contracts/test/v2/VaultFeeOnTransfer.t.sol`).

3.3.4 Unused `perpClearing` in `AccountLedger`

Severity: Informational

Context: AccountLedger.sol#L17

Description: The `perpClearing` variable is defined in `SpotSettlement` but never used. Since the contract is not upgradeable, this placeholder serves no practical purpose. It introduces dead code that may confuse reviewers and complicate maintenance.

Recommendation: Remove `perpClearing` and its associated roles to maintain the code clean.

Clove: Fixed in PR 20 (commit 78454b01).

Cantina Managed: Fix verified. The unused `perpClearing` role/state in `AccountLedger` was removed, including the dead `PERP_CLEARING_ROLE` references.