Steakhouse Financial: Vault v2 Supervisor

Security Review

Cantina Solo review by:
Alireza Arjmand, Lead Security Researcher

February 23, 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

A security review is 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 the review 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 a 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

Steakhouse builds transparent, efficient, and accessible financial primitives to power the next generation of capital markets on public blockchains.

From Feb 10th to Feb 11th the security researchers conducted a review of vault-v2-supervisor on commit hash 34b2ec29. A total of **11** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	3	2	1
Gas Optimizations	0	0	0
Informational	8	7	1
Total	**11**	**9**	**2**

2.1 Scope

The security review had the following components in scope for vault-v2-supervisor on commit hash 34b2ec29:

```
├── src
│   └── VaultV2Supervisor.sol
└── test
    └── VaultV2Supervisor.t.sol
```

2.2 Trust assumptions

This system relies on several strong trust assumptions around privileged roles and vault configuration.

2.2.1 Guardian Role

The guardian set is fully trusted. If an incorrect or malicious address is added as a guardian, the system can be irreversibly compromised. Guardians are expected to act honestly and defensively. In particular, while the owner may become malicious, guardians are assumed to monitor and intervene where possible to prevent harmful actions.

2.2.2 Owner and Sentinel Control

For vaults owned by the `VaultV2Supervisor`, guardians are always able to configure the supervisor as the vault's sentinel and retain access to revoke functionality. This ensures an emergency control path remains available.

For vaults not owned by the `VaultV2Supervisor`, this assumption no longer holds. The vault owner can remove the supervisor from the sentinels list, thereby blocking its ability to act as a sentinel or use revoke-related functionality. In such configurations, safety depends entirely on the vault owner's behavior.

2.2.3 Curator Role and Supervisor Control

The curator role on vaults is fully controlled by the supervisor. While curators are not expected to be able to directly steal funds without going through timelocked mechanisms, they can influence vault parameters in ways that materially affect usability.

Specifically, a curator can modify absolute caps and relative caps in a way that renders the vault effectively unusable for users. This is considered within their authority and is therefore a trust assumption rather than a bug.

Additionally, when the contract only has sentinel access, it cannot call `decreaseRelativeCap`, `decreaseAbsoluteCap`, or `deallocate`. This limits the blast radius of sentinel-only permissions.

2.2.4 Underlying Vault Configuration

All underlying vaults are assumed to be correctly configured according to the intended Morpho Vault V2 design and to have been properly audited. Misconfiguration or unsafe customizations at the underlying vault layer may invalidate the security assumptions of this system.

2.3 Reviewer Statement

In this solo review, Cantina reviewed Steakhouse's `VaultV2Supervisor` contract. The team was responsive throughout the process and addressed feedback promptly and thoroughly. Based on the scope of this review and the fixes implemented, the codebase is in good condition and suitable for deployment.

3 Overview

`VaultV2Supervisor` is a governance wrapper intended to own one or more Morpho Vault V2 instances and provide a supervisory control plane with a global timelock and per-vault guardian registry. It implements two-step supervisor ownership transfer, a guardian allowlist for vault owners, timelocked scheduling and execution for specific supervisor actions, and a guardian proxy that can forward vault-level `revoke(bytes)` calls.

The supervisor exposes:

- Immediate owner actions to manage vault metadata and roles (e.g., curator, sentinels, name/symbol), set skim recipient on adapters, and manage guardian allowlisting.
- Timelocked actions (`setOwner`, `removeGuardian`) that must be submitted, delayed, and then executed, with the ability for guardians or the owner to revoke pending actions.
- Vault tracking helpers (vaults with guardians, owned vs. non-owned).
- A guardian proxy for Vault V2 timelock revocations, enabled when the supervisor is set as a vault sentinel.

Vault V2 itself distinguishes owner, curator, sentinel, and allocator roles. Owner actions are immediate; curator actions are generally timelocked via the vault's `submit/timelocked` mechanism; sentinel and allocator actions are immediate within their respective permission scopes.

3.1 Validating Supervisor Use Cases Against Vault V2 Capabilities

Assuming the system is configured safely and all guardians are trusted, the Supervisor owner's control path across the Supervisor $\rightarrow$ Vault V2 stack is as follows.

3.1.1 Supervisor layer (direct control)

- Immediate owner actions: `setCurator`, `setName`, `setSymbol`, `addSentinel`, `removeSentinel`, `setSkimRecipient`, `setAllowedVaultOwner`, `addGuardian`, and two-step `transferSupervisorOwnership`.
- Timelocked and guardian-revocable actions: `setOwner` (vault ownership transfer) and `removeGuardian`, via `submit → wait → execute`. Guardians or the owner can veto via `revoke`.

3.1.2 Vault V2 layer (via supervisor as owner/curator)

- Owner actions (immediate at vault): `setCurator`, `setIsSentinel`, `setName`, `setSymbol`.
- Since curator/sentinel can be updated at supervisor's will, supervisor does not need to wait behind a timelock for below actions (immediate at vault):
  `submit` (curator), `revoke` (curator or sentinel), `decreaseAbsoluteCap` (curator or sentinel), `decreaseRelativeCap` (curator or sentinel), `deallocate` (allocator or sentinel but updating sentinel is faster).
- Curator actions (vault timelocked via `submit`):
  `setIsAllocator`, `setReceiveSharesGate`, `setSendSharesGate`, `setReceiveAssetsGate`, `setSendAssetsGate`, `setAdapterRegistry`, `addAdapter`, `removeAdapter`, `increaseTimelock`, `decreaseTimelock`, `abdicate`, `setPerformanceFee`, `setManagementFee`, `setPerformanceFeeRecipient`, `setManagementFeeRecipient`, `increaseAbsoluteCap`, `increaseRelativeCap`, `setForceDeallocatePenalty`.
- Allocator/sentinel actions are immediate at vault, but for the supervisor to be able to change the allocator, malicious `setIsAllocator` needs to wait behind a timelock:
  `allocate`, `setLiquidityAdapterAndData` and `setMaxRate`.
  The allocator role itself is granted only via the timelocked curator action `setIsAllocator`.

3.1.3 Guardian veto path

- Guardians can veto supervisor-level timelocks (`setOwner`, `removeGuardian`) through the supervisor's `revoke`.
- Guardians can veto vault-level timelocks only if the supervisor is a vault sentinel; this is enabled via packing `setSupervisorAsSentinel` with `supervisor.revoke(vault, data)` to forward the veto to Vault V2.

4 Findings

4.1 Low Risk

4.1.1 Timelocked Action Can Be Made Hard to Revoke and Invisible to `isGuardianBeingRemoved` via Trailing Calldata

Severity: Low Risk

Context: VaultV2Supervisor.sol#L173-L205

Description: Timelocked actions are keyed by the exact `bytes data` in `executableAt[data]`. Since Solidity ignores trailing calldata or dirty bytes for decoding but keeps it in `msg.data`, the owner can submit a valid action with extra trailing bytes. That action executes normally, but helper functions that reconstruct canonical calldata cannot find and revoke it.

This breaks:

- `revokeGuardianRemoval` and `revokeVaultOwnerChange` (they encode clean calldata).
- `isGuardianBeingRemoved` (it also encodes clean calldata, so it returns false).

> **Note:** the action is still revokable via the generic `revoke(bytes data)` if the exact submitted `data` (including the extra bytes) is known.

Proof of Concept:

```
function test_ImpossibleRevokeGuardianRemoval() public {
    supervisor.addGuardian(address(vault), GUARDIAN);

    bytes memory data = abi.encodePacked(abi.encodeWithSelector(VaultV2Supervisor.remove
    ↪  Guardian.selector, vault, GUARDIAN), hex'11'); // Appending random data at the
    ↪  end of the calldata
    supervisor.submit(data);

    vm.startPrank(GUARDIAN);
    vm.expectRevert();
    supervisor.revokeGuardianRemoval(vault, GUARDIAN);
    vm.stopPrank();

    vm.warp(block.timestamp + TIMELOCK);

    address[] memory guardiansBefore = supervisor.getGuardians(address(vault));

    (bool success, ) = address(supervisor).call(data);
    require(success);

    address[] memory guardiansAfter = supervisor.getGuardians(address(vault));

    vm.assertEq(guardiansAfter.length, guardiansBefore.length - 1); // Number of
    ↪  guardians reduced
}
```

Recommendation: Make sure the data that is being received for submission does not include dirty bytes or trailing calldata.

Steakhouse: Fixed in commit b997d7de.

Alireza Arjmand: The Steakhouse team mitigated this issue by validating, at submission time, that the calldata has the expected length and contains no dirty bytes, using the `_decodeCanonicalTwoAddressCalldata` function.

4.1.2 Guardians Must Revoke Timelocked Actions One-by-One

Severity: Low Risk

Context: VaultV2Supervisor.sol#L136-L171

Description: A malicious owner can submit many timelocked actions, forcing guardians to revoke them individually. Since revocation is per exact calldata entry, guardians pay $\mathcal{O}(n)$ gas to clean up n scheduled actions, matching the owner's griefing budget and potentially making defense impractical during an incident.

Recommendation: Add an $\mathcal{O}(1)$ circuit breaker, for example a pause mechanism or global "revoke all pending actions" switch (optionally per vault), so guardians can halt execution without needing to revoke every scheduled item individually.

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes. There is now a single canonical calldata that corresponds to a timelocked action, so guardians only need to revoke one calldata per action if required.

4.1.3 Guardian Tracking Can Be DoSed by Adding Many Vaults

Severity: Low Risk

Context: VaultV2Supervisor.sol#L286-L314

Description: An allowlisted vault owner (or the supervisor owner) can call `addGuardian` across a large number of vault addresses, growing `_vaults` and `_guardians`. This can make `getVaults`, `getOwnedVaults`, `getNonOwnedVaults`, and per-vault guardian enumeration increasingly expensive and potentially unusable for onchain callers.

Recommendation: Consider whitelisting vaults instead of vault owners so that number of vaults can't suddenly grow.

Steakhouse: This behavior is intentional, and we accept the associated trade-offs.

Alireza Arjmand: Acknowledged.

4.2 Informational

4.2.1 Immutable Naming Convention Deviation

Severity: Informational

Context: VaultV2Supervisor.sol#L79

Description: The immutable variable `timelock` does not follow Solidity's recommended naming convention for constants and immutables, which should use `UPPER_CASE_WITH_UNDERSCORES` as specified in the Solidity style guide: `https://docs.soliditylang.org/en/v0.8.10/style-guide.html#constants`.

Recommendation: Rename `timelock` to `TIMELOCK` to align with Solidity conventions.

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes.

4.2.2 Missing `onlyOwnerOrGuardian` Modifier

Severity: Informational

Context: VaultV2Supervisor.sol#L162

Description: The contract repeats the same access check inline instead of using a modifier:

```
_guardians[vault].contains(msg.sender) || msg.sender == owner
```

This pattern appears in `revoke` and `_revokeKnownVault`. The latter is called by `revokeGuardianRemoval` and `revokeVaultOwnerChange`, so placing the check on those external entrypoints would improve readability and separation of concerns.

Recommendation: Add an `onlyOwnerOrGuardian(address vault)` modifier and apply it to relevant functions, especially the external callers of `_revokeKnownVault`.

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes.

4.2.3 Non-Descriptive Variable Names in Decoding

Severity: Informational

Context: VaultV2Supervisor.sol#L145

Description: In `submit`, decoded variables use abbreviated names:

```
(address v, address new0) = abi.decode(...)
```

Short names reduce readability and make the intent less clear, especially in security-sensitive logic.

Recommendation: Use descriptive names such as:

```
(address _vault, address _newOwner)
```

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes.

4.2.4 `_selectorUsesVault` Includes Non-Timelocked Selector

Severity: Informational

Context: VaultV2Supervisor.sol#L394

Description: `_selectorUsesVault` includes `removeSentinel`, but `removeSentinel` is not timelocked and does not go through the `submit` flow. This makes the selector list misleading.

Recommendation: Remove `removeSentinel` from `_selectorUsesVault` (and any related selector lists used for timelock context).

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes.

4.2.5 Missing Minimum Bound for Timelock

Severity: Informational

Context: VaultV2Supervisor.sol#L107

Description: The constructor check `require(timelock_ > 0, InvalidTimelock());` only ensures the value is nonzero but does not enforce that it is practically safe. Extremely small values could undermine the purpose of a timelock.

Recommendation: Introduce a minimum bound constant, e.g. `MINIMUM_TIMELOCK`, and validate against it to ensure the configured delay is meaningful.

Steakhouse: We don't consider this an issue, as we will only deploy one supervisor per chain, and the value is immutable.

Alireza Arjmand: Acknowledged. Provided the deployment is properly verified, no further changes are required.

4.2.6 Duplicate Revoke Logic in `revoke` and `_revokeKnownVault`

Severity: Informational

Context: VaultV2Supervisor.sol#L162-L170

Description: `revoke(bytes calldata data)` duplicates the same core logic implemented in `_revokeKnownVault` (permission check, timelock existence check, clearing `scheduledNewOwner` when relevant, clearing `executableAt`, emitting event). This is unnecessary duplication and slightly increases maintenance risk.

Recommendation: Refactor `revoke(bytes)` to reuse `_revokeKnownVault` (or a shared internal helper) so the revoke logic lives in one place.

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes.

4.2.7 Function Order Does Not Follow Solidity Convention

Severity: Informational

Context: VaultV2Supervisor.sol#L3

Description: The contract does not follow Solidity's recommended function ordering, which can make navigation and review harder. The suggested order is documented here: `https://docs.soliditylang.org/en/latest/style-guide.html#order-of-functions`.

Recommendation: Reorder functions to follow Solidity's standard structure to improve readability and consistency.

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes.

4.2.8 Cannot Cancel Pending Supervisor Ownership Transfer

Severity: Informational

Context: VaultV2Supervisor.sol#L115

Description: Once `transferSupervisorOwnership(newOwner)` sets `pendingSupervisorOwner`, there is no direct way for the current owner to cancel it. To "cancel," the owner must overwrite it by proposing another address, which is clunky and can be error-prone.

Recommendation: Add an explicit cancel function (or allow `transferSupervisorOwnership(address(0))`) to clear `pendingSupervisorOwner`.

Steakhouse: Fixed in PR 2.

Alireza Arjmand: Verified the fixes. It is now possible to call `transferSupervisorOwnership` with `address(0)` as the calldata to cancel an ongoing transfer process.

5 Appendix

5.1 Fuzz Test Added for Submit Workflow

```
  diff --git a/test/VaultV2Supervisor.t.sol b/test/VaultV2Supervisor.t.sol
  index 5490a88..4a5158b 100644
-  -- a/test/VaultV2Supervisor.t.sol
+  ++ b/test/VaultV2Supervisor.t.sol
  @@ -478,6 +478,142 @@ contract VaultV2SupervisorTest is Test {
           assertFalse(supervisor.isVaultSupervised(address(0xB0B)));
       }

+      function testFuzz_SubmitRevokeWorkflow_AlwaysClearsPendingState(uint256 seed,
↪  uint8 steps) public {
+          steps = uint8(bound(steps, 6, 30));
+
+          address guardianA = address(0xBEE1);
+          address guardianB = address(0xBEE2);
+          supervisor.addGuardian(address(vault), guardianA);
+          supervisor.addGuardian(address(vault), guardianB);
+
+          bytes32 state = keccak256(abi.encode(seed));
+
+          for (uint256 i; i < steps; ++i) {
+              state = keccak256(abi.encode(state, i));
+              bool ownerWorkflow = (uint256(state) & 1) == 1;
+              bool revokeByGuardian = ((uint256(state) >> 1) & 1) == 1;
+              bool useSpecializedRevoke = ((uint256(state) >> 2) & 1) == 1;
+
+              if (ownerWorkflow) {
+                  address candidate = _ownerCandidateFromState(state);
+                  bytes memory data =
↪  abi.encodeWithSelector(VaultV2Supervisor.setOwner.selector, vault, candidate);
+                  bytes memory oversized = bytes.concat(data, new bytes((uint256(state)
↪  % 8) + 1));
+                  bytes memory dirtyVault = abi.encodePacked(
+                      VaultV2Supervisor.setOwner.selector,
+                      _dirtyAddressWord(address(vault)),
+                      bytes32(uint256(uint160(candidate)))
+                  );
+                  bytes memory dirtyOwner = abi.encodePacked(
+                      VaultV2Supervisor.setOwner.selector,
+                      bytes32(uint256(uint160(address(vault)))),
+                      _dirtyAddressWord(candidate)
+                  );
+
+                  vm.expectRevert(VaultV2Supervisor.InvalidAmount.selector);
+                  supervisor.submit(oversized);
+                  vm.expectRevert(VaultV2Supervisor.InvalidAmount.selector);
+                  supervisor.submit(dirtyVault);
+                  vm.expectRevert(VaultV2Supervisor.InvalidAmount.selector);
+                  supervisor.submit(dirtyOwner);
+
+                  uint256 submitTime = block.timestamp;
+                  supervisor.submit(data);
+
+                  assertEq(supervisor.executableAt(data), submitTime + TIMELOCK);
+                  assertEq(supervisor.scheduledNewOwner(address(vault)), candidate);
+                  assertTrue(supervisor.isOwnershipChanging(address(vault)));
+
+                  vm.expectRevert(VaultV2Supervisor.DataNotTimelocked.selector);
+                  supervisor.revoke(oversized);
+
+                  if (revokeByGuardian) vm.prank(guardianA);
+
+                  if (useSpecializedRevoke) {
```

```
+                     supervisor.revokeVaultOwnerChange(vault);
+                 } else {
+                     supervisor.revoke(data);
+                 }
+
+                 assertEq(supervisor.executableAt(data), 0);
+                 assertEq(supervisor.scheduledNewOwner(address(vault)), address(0));
+                 assertFalse(supervisor.isOwnershipChanging(address(vault)));
+
+                 vm.expectRevert(VaultV2Supervisor.DataNotTimelocked.selector);
+                 supervisor.setOwner(vault, candidate);
+             } else {
+                 address guardianToRemove = ((uint256(state) >> 3) & 1) == 1 ?
↪  guardianA : guardianB;
+                 bytes memory data =
+                     abi.encodeWithSelector(VaultV2Supervisor.removeGuardian.selector,
↪  vault, guardianToRemove);
+                 bytes memory oversized = bytes.concat(data, new bytes((uint256(state)
↪  % 8) + 1));
+                 bytes memory dirtyVault = abi.encodePacked(
+                     VaultV2Supervisor.removeGuardian.selector,
+                     _dirtyAddressWord(address(vault)),
+                     bytes32(uint256(uint160(guardianToRemove)))
+                 );
+                 bytes memory dirtyGuardian = abi.encodePacked(
+                     VaultV2Supervisor.removeGuardian.selector,
+                     bytes32(uint256(uint160(address(vault)))),
+                     _dirtyAddressWord(guardianToRemove)
+                 );
+
+                 vm.expectRevert(VaultV2Supervisor.InvalidAmount.selector);
+                 supervisor.submit(oversized);
+                 vm.expectRevert(VaultV2Supervisor.InvalidAmount.selector);
+                 supervisor.submit(dirtyVault);
+                 vm.expectRevert(VaultV2Supervisor.InvalidAmount.selector);
+                 supervisor.submit(dirtyGuardian);
+
+                 uint256 submitTime = block.timestamp;
+                 supervisor.submit(data);
+
+                 assertEq(supervisor.executableAt(data), submitTime + TIMELOCK);
+                 assertTrue(supervisor.isGuardianBeingRemoved(address(vault),
↪  guardianToRemove));
+
+                 vm.expectRevert(VaultV2Supervisor.DataNotTimelocked.selector);
+                 supervisor.revoke(oversized);
+
+                 if (revokeByGuardian) vm.prank(guardianA);
+
+                 if (useSpecializedRevoke) {
+                     supervisor.revokeGuardianRemoval(vault, guardianToRemove);
+                 } else {
+                     supervisor.revoke(data);
+                 }
+
+                 assertEq(supervisor.executableAt(data), 0);
+                 assertFalse(supervisor.isGuardianBeingRemoved(address(vault),
↪  guardianToRemove));
+                 assertTrue(_contains(supervisor.getGuardians(address(vault)),
↪  guardianToRemove));
+
+                 vm.expectRevert(VaultV2Supervisor.DataNotTimelocked.selector);
+                 supervisor.removeGuardian(vault, guardianToRemove);
+             }
+
+             address[] memory guardians = supervisor.getGuardians(address(vault));
```

```
+             assertEq(guardians.length, 2);
+             assertTrue(_contains(guardians, guardianA));
+             assertTrue(_contains(guardians, guardianB));
+
+             address[] memory vaults = supervisor.getVaults();
+             assertEq(vaults.length, 1);
+             assertEq(vaults[0], address(vault));
+             assertEq(vault.owner(), address(supervisor));
+         }
+     }
+
+     function _ownerCandidateFromState(bytes32 state) internal view returns (address
 ↪  candidate) {
+         candidate = address(uint160(uint256(state)));
+         if (candidate == address(0) || candidate == address(supervisor)) {
+             candidate = address(uint160(uint256(keccak256(abi.encode(state,
 ↪  "owner-candidate")))));
+         }
+         if (candidate == address(0) || candidate == address(supervisor)) {
+             candidate = address(0x1111111111111111111111111111111111111111);
+         }
+     }
+
+     function _dirtyAddressWord(address account) internal pure returns (bytes32) {
+         return bytes32(uint256(uint160(account)) | (uint256(1) << 200));
+     }
+
      function _contains(address[] memory list, address account) internal pure returns
      ↪  (bool) {
          for (uint256 i; i < list.length; ++i) {
              if (list[i] == account) return true;
```