Sprinter Credit: Remote Collateral Contracts

Security Review

Cantina Managed review by:
Slowfi, Lead Security Researcher
Kankodu, Security Researcher

April 7, 2026

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
2.1 Scope . . . 3

3 Findings **4**
3.1 High Risk . . . 4
3.1.1 Controller settlement uses asset based `ERC4626` flows for share accounted collateral 4
3.1.2 Arbitrary `wrapData` in `EscrowHelper` Allows Attacker to Steal User Funds . . . 5
3.2 Medium Risk . . . 7
3.2.1 Borrow validations in `StashCreditHub` enforce tighter limits . . . 7
3.2.2 Liquidation rounding can seize more collateral than the intended reward value . . . 7
3.2.3 Attacker Can DoS Victim's Collateral Deposits by Filling Their Collateral Array . . . 8
3.2.4 `previewRedeem` in `ChainlinkOracle` Reverts for ERC7540 Async Vaults . . . 10
3.2.5 Liquidation outcome is non monotonic with respect to the requested amount . . . 10
3.3 Low Risk . . . 13
3.3.1 Zero amount withdrawals can arbitrarily advance the global withdrawal nonce . . . 13
3.3.2 Revocation finalization and borrowing remain simultaneously valid at the revoke boundary . . . 13
3.3.3 Zero liquidation bonus weakens the collateral initialization guard . . . 13
3.3.4 Oracle configuration cannot be symmetrically cleared across pricing modes . . . 14
3.3.5 `ERC4626` asset identifier validation is deferred to runtime instead of configuration time 14
3.3.6 Static price configuration does not bound decimals and can break oracle valuation . . . 15
3.3.7 Liquidation order accepts origins with zero borrower deposit . . . 15
3.3.8 Registered collateral can become depositable before it is priceable . . . 16
3.3.9 Zero amount repay can inflate borrower fees through repeated credit line refresh . . . 16
3.3.10 Pause mechanism not strongly defined . . . 17
3.3.11 `maxWithdraw` can overestimate removable collateral for origins that share a global asset . . . 17
3.3.12 `EscrowVault` Uses Default `decimalOffset` of Zero, Leaving Inflation Attack Possible 18
3.3.13 Suggestion on How to Protect `repayWithPermit` from Griefing . . . 18
3.4 Informational . . . 18
3.4.1 Add `EscrowHelper` usage documentation . . . 18
3.4.2 Use of assert on production code . . . 19
3.4.3 Signed deposit path accepts local origin tuples . . . 19
3.4.4 Escrow design lacks a recovery path for controller lock failures . . . 20
3.4.5 Cross chain deposit signing depends on offchain finality handling . . . 20
3.4.6 Admin liquidity withdrawal can desynchronize internal accounting invariants . . . 21
3.4.7 Unconfigured domain receivers can leave liquidation claims pending in the helper queue . . . 21
3.4.8 Liquidator helper cannot handle empty claim sets and reverts instead . . . 22
3.4.9 Missing Events on State Changing Functions . . . 22
3.4.10 Unused Custom Errors in `CreditHubBase` . . . 22
3.4.11 `liquidationBonus >= 0` Check Is Always True . . . 23
3.4.12 `onlyOperatorOrBorrower` Modifier Name Is Misleading . . . 23
3.4.13 Missing `debtors` Array Length View Function . . . 23
3.4.14 `drawCredit` transfer credit asset before enforcing the hub invariants . . . 23
3.4.15 Unbounded oracle rotation path . . . 24
3.4.16 Liquidation math does not account for collateral unit granularity and can break partial liquidation behavior . . . 24

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

Sprinter Credit offers programmable credit infrastructure for the onchain economy. Any app, neobank, wallet, card program or agent - plugs in via one API and gets a full credit stack. DeFi in the back, fintech UX in the front.

From Mar 1st to Mar 15th the Cantina team conducted a review of remote-collateral-contracts on commit hash 404c97a2. The team identified a total of **36** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	5	4	1
Low Risk	13	11	2
Gas Optimizations	0	0	0
Informational	16	11	5
Total	**36**	**28**	**8**

2.1 Scope

The security review had the following components in scope for remote-collateral-contracts on commit hash 404c97a2:

```
contracts
├── ChainlinkOracle.sol
├── CreditHub.sol
├── CreditHubBase.sol
├── CreditHubController.sol
├── EndOfMonthTimestamps.sol
├── Escrow.sol
├── EscrowController.sol
├── EscrowHelper.sol
├── EscrowLocal.sol
├── EscrowVault.sol
├── StashCreditHub.sol
├── liquidation
│   ├── Liquidator.sol
│   ├── Preprocessor.sol
│   ├── PreprocessorBase.sol
│   ├── PreprocessorLocal.sol
│   └── Processor.sol
├── operator
│   ├── ExclusiveOperator.sol
│   └── Operator.sol
└── utils
    └── ERC7201Helper.sol
```

3 Findings

3.1 High Risk

3.1.1 Controller settlement uses asset based ERC4626 flows for share accounted collateral

Severity: High Risk

Context: EscrowController.sol#L46-L50

Description: The function `withdraw` from contract `EscrowController` uses the signed `amount` as an asset amount when calling the ERC4626 withdrawal flow, while the corresponding collateral accounting and escrow locking logic operate in share units.

This creates a unit mismatch for variable rate escrows such as `EscrowVault`. In the remote flow, the same signed `amount` can be consumed through `unlock` as shares or through `withdraw` as assets. When the vault exchange rate changes, these two paths no longer consume the same quantity of shares for the same signed value.

As a result, controller settlement can become inconsistent with hub side collateral accounting. If the vault exchange rate increases, withdrawing a fixed asset amount burns fewer shares than the hub decrements, which can leave locked shares and residual value in the escrow that can no longer be released through the hub flow. If the vault exchange rate decreases, withdrawing a fixed asset amount burns more shares than the hub decrements, which can leave more collateral recorded in the hub than remains locked in the escrow.

The same pattern also affects local controller settlement paths that use ERC4626 asset based withdrawals for collateral that is tracked in shares. In practice, this can cause stranded value or phantom collateral if non one to one vault escrows are allowed as collateral.

Proof Of Concept: A user deposits and locks vault shares through `EscrowVault`, and the hub records the credited collateral in shares.

Later, the vault exchange rate increases because the escrow balance grows without additional shares being minted.

When the user withdraws a fixed asset amount through the controller flow, the ERC4626 withdrawal burns fewer shares than the asset amount. However, the hub still reduces `originDeposits` and `collateral` by the full requested amount.

After that withdrawal:

- The escrow still has locked shares remaining for the user.
- The hub tracks a smaller remaining origin deposit than the actual locked share position.
- The residual locked value cannot be fully released through the normal hub withdrawal flow.

This demonstrates that controller settlement and hub accounting diverge once the vault ratio changes.

```
it("creates stranded locked value after rebase up withdraw flow", async function () {
  const {
    creditHub,
    controller,
    user1,
    receiver,
    testUSDC,
    escrowVault,
    ORIGIN_DOMAIN_ID,
    ORIGIN_ASSET_ID,
    GLOBAL_ASSET_ID,
  } = await networkHelpers.loadFixture(deployAll);

  const depositAmount = 200_000000n;
  const mintAmount = 200_000000n;
  const withdrawAmount = 100_000000n;

  await testUSDC.mint(user1, 1000_000000n);
  await testUSDC.connect(user1).approve(escrowVault, depositAmount);
```

```
  const [, sharesLocked] = await escrowVault.connect(user1).depositAndLock.staticCall(
    depositAmount,
    user1,
    controller,
  );
  await escrowVault.connect(user1).depositAndLock(depositAmount, user1, controller);
  await creditHub.connect(controller).deposit(user1, ORIGIN_DOMAIN_ID, ORIGIN_ASSET_ID,
  ↪  sharesLocked);

  // Simulate vault gains so withdrawing a fixed asset amount burns fewer shares.
  await testUSDC.mint(escrowVault, mintAmount);
  const burnedShares = await escrowVault.previewWithdraw(withdrawAmount);
  expect(burnedShares).to.be.lessThan(withdrawAmount);

  await creditHub.connect(controller).withdraw(
    user1,
    user1,
    ORIGIN_DOMAIN_ID,
    ORIGIN_ASSET_ID,
    addressToBytes32(receiver.address),
    withdrawAmount,
  );
  await escrowVault.connect(controller).withdraw(withdrawAmount, receiver, user1);

  const lockedRemaining = sharesLocked - burnedShares;
  const hubRemaining = sharesLocked - withdrawAmount;

  expect(await escrowVault.locked(user1)).to.equal(lockedRemaining);
  expect(await creditHub.collateral(user1, GLOBAL_ASSET_ID)).to.equal(hubRemaining);
  expect(await creditHub.originDeposits(user1, ORIGIN_DOMAIN_ID,
  ↪  ORIGIN_ASSET_ID)).to.equal(hubRemaining);
  expect(lockedRemaining).to.be.greaterThan(hubRemaining);

  const strandedShares = lockedRemaining - hubRemaining;
  expect(strandedShares).to.be.greaterThan(0n);
  expect(await escrowVault.previewRedeem(strandedShares)).to.be.greaterThan(0n);

  await expect(
    creditHub.connect(controller).withdraw(
      user1,
      user1,
      ORIGIN_DOMAIN_ID,
      ORIGIN_ASSET_ID,
      addressToBytes32(receiver.address),
      lockedRemaining,
    )
  ).to.be.revertedWithCustomError(creditHub, "InsufficientOriginDeposit");
});
```

Recommendation: Consider to keep controller settlement fully share based for variable rate escrows. Using ERC4626 `redeem` semantics instead of asset based `withdraw` semantics would align controller execution with the share based accounting used by escrow locking and hub collateral tracking.

Consider also reviewing all local and remote withdrawal and claim paths to ensure the same unit is used consistently end to end for non one to one escrows.

Sprinter Tech: Fixed in PR 28 by using redeem instead of withdraw.

Cantina Managed: Fix verified.

3.1.2 Arbitrary `wrapData` in `EscrowHelper` Allows Attacker to Steal User Funds

Severity: High Risk

Context: EscrowHelper.sol#L47-L59

Description: `wrapAllDepositAndLock` accepts a caller supplied `wrapData` bytes payload and executes it via `address(token).functionCall(wrapData)`. Users who have approved the `EscrowHelper` contract (as required by the `pullWrapDepositAndLock` flow) can have their full allowance drained in a single call.

An attacker calls `wrapAllDepositAndLock` with a malicious `wrapData` encoding `token.transferFrom(victim, attacker, amount)`. Because the victim already approved `EscrowHelper`, the call succeeds and transfers the victim's tokens directly to the attacker.

Proof of Concept:

```
pragma solidity 0.8.28;

import "forge-std/Test.sol";
import "../src/EscrowHelper.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract Token is ERC20 {
    constructor() ERC20("Mock", "MOCK") {}
    function mint(address to, uint256 amount) external { _mint(to, amount); }
}

contract DummyEscrow is IEscrow {
    function unlock(uint256 amount, address user) external {}
    function depositAndLock(uint256, address, address) external pure returns (uint256,
    ↪  uint256) {
        return (0, 0);
    }
}

contract EscrowHelperExploitTest is Test {
    EscrowHelper helper;
    Token token;
    DummyEscrow escrow;

    address victim   = makeAddr("victim");
    address attacker = makeAddr("attacker");

    function setUp() public {
        helper = new EscrowHelper();
        token  = new Token();
        escrow = new DummyEscrow();

        token.mint(victim, 1000 ether);

        vm.prank(victim);
        token.approve(address(helper), type(uint256).max);
    }

    function testExploit() public {
        uint256 victimBalance = token.balanceOf(victim);

        bytes memory maliciousWrapData = abi.encodeWithSelector(
            token.transferFrom.selector, victim, attacker, victimBalance
        );

        vm.prank(attacker);
        helper.wrapAllDepositAndLock(escrow, token, attacker, attacker, token,
        ↪  maliciousWrapData);

        assertEq(token.balanceOf(victim), 0);
        assertEq(token.balanceOf(attacker), victimBalance);
    }
}
```

Recommendation: Restrict `wrapData` to only call a known, whitelisted wrap contract and function selector,

or remove the `pull*` functions entirely to make sure no one approves this contract.

Sprinter Tech: Fixed in PR 28 by removing pull functions.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Borrow validations in `StashCreditHub` enforce tighter limits

Severity: Medium Risk

Context: StashCreditHub.sol#L53-L60

Description: The function `_openCreditLine` from contract `StashCreditHub` withdraws liquidity from `STASH_POOL` before executing the base credit line logic.

In the base implementation, borrow validations such as utilization checks and per-block borrow limits assume that liquidity has not yet been transferred out when the validations run. However, in the `StashCreditHub` implementation, liquidity is already reduced when `STASH_POOL.withdraw(address(this), amount)` is executed before calling `super._openCreditLine`.

As a result, the validation logic evaluates utilization and borrow limits using a reduced liquidity value. This causes the checks to be stricter than the configured parameters. In practice, borrows that would otherwise be valid according to the intended configuration can revert because the validation formulas operate on liquidity that already reflects the pending withdrawal.

This creates a discrepancy between the configured borrow limits and the effective limits enforced by the contract.

Recommendation: Consider to ensure that borrow limit validations are executed using the withdraw liquidity state. This can be achieved by performing the validation logic before calling `STASH_POOL.withdraw`, or by implementing a stash-specific borrowing flow that evaluates the checks against the pool liquidity before reducing it.

Sprinter Tech: Fixed in PR 28 by transferring liquidity out of the hub and assuming so before validation.

Cantina Managed: Fix verified.

3.2.2 Liquidation rounding can seize more collateral than the intended reward value

Severity: Medium Risk

Context: CreditHubBase.sol#L564-L566

Description: The function `_liquidateOrigin` from contract `CreditHubBase` uses upward rounding when converting reward value into seized collateral amount.

More specifically, `remainingRewardValue` is rounded up, `liquidateValue` is bounded by `depositValue`, and `liquidateAmount` is then computed with upward rounding as well. This makes the seized collateral amount conservative in favor of the liquidator. However, the reward budget progression is not updated with the same level of conservatism, so the collateral seizure and the remaining reward accounting are not fully symmetric.

As a result, liquidation can seize collateral whose value is greater than the value implied by the repaid debt plus the configured liquidation bonus. The effect is more visible for small value liquidations and for collateral with coarse units or high unit value, where a very small repayment may still round to one full collateral unit.

In addition, the fallback liquidation loop in `liquidate` continues iterating across origins without stopping on a no progress step. This can make the behavior less controlled when several origins are processed under the same rounded reward budget.

This causes the borrower to lose more collateral than intended by the liquidation parameters.

Recommendation: Consider to make the liquidation accounting use consistent conservative rounding across reward computation, seized amount computation, and remaining reward updates. Consider to round the seized amount down or cap it strictly to the exact reward budget, and consider to stop the

fallback loop when a no progress step is reached. Consider also enforcing a minimum liquidation amount if needed to reduce dust driven rounding effects.

Sprinter Tech: Fixed in PR 28 by not rewarding liquidator in case the upward rounded reward value is smaller than value of a single collateral unit.

Cantina Managed: Fix verified.

3.2.3 Attacker Can DoS Victim's Collateral Deposits by Filling Their Collateral Array

Severity: Medium Risk

Context: CreditHubBase.sol#L405

Description: `CreditHub.deposit` adds the deposited asset's `globalAssetId` to `$.collaterals[receiver]` and enforces a `MAX_GLOBAL_ASSETS` cap. Because any caller can call `EscrowLocal.depositAndLock(amount = 1 wei, receiver = victim, controller = creditHubController)`, an attacker can deposit dust amounts on behalf of a victim and fill all of their collateral slots before the victim deposits their intended asset.

Once the victim's `$.collaterals` array is at the limit, any further deposit attempt from the victim reverts with `TooManyGlobalAssets()`. The attacker gains nothing financially, but this can be used to grief specific accounts. Additionally, filling the array to the maximum means every health-factor check for the victim burns the maximum amount of gas.

Proof of Concept: Here's the proof of concept with `MAX_GLOBAL_ASSETS = 2` (patched in `CreditHubBase` + only showing local deposits for simplicity).

```
// SPDX-License-Identifier: LGPL-3.0-only
pragma solidity 0.8.28;

import {Test, console} from "forge-std/Test.sol";
import {CreditHub} from "contracts/CreditHub.sol";
import {CreditHubBase} from "contracts/CreditHubBase.sol";
import {CreditHubController} from "contracts/CreditHubController.sol";
import {EscrowLocal} from "contracts/EscrowLocal.sol";
import {ValueOracle} from "contracts/ValueOracle.sol";
import {IOracle} from "contracts/interfaces/IOracle.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract MockERC20 is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbol) {}

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

// Attack vector:
//   MAX_GLOBAL_ASSETS = 2 (patched in CreditHubBase for POC clarity).
//   Attacker deposits 1 wei of wBTC and wstETH with receiver = victim, filling both
↪  slots.
//   Victim then cannot deposit their intended collateral (ETH) — TooManyGlobalAssets().
//
//   Call chain per attacker deposit:
//     EscrowLocal.depositAndLock(1, victim, creditHubController)
//       → EscrowLocal._lock → CreditHubController.depositLocal
//         → CreditHub.deposit  ← $.collaterals[victim].add(globalAssetId)
contract DOSBorrowerCollateralLimitPOC is Test {
    address admin    = makeAddr("admin");
    address attacker = makeAddr("attacker");
    address victim   = makeAddr("victim");

    CreditHub           creditHub;
    CreditHubController creditHubController;
```

```
    ValueOracle         oracle;

    MockERC20 usdc;   // hub borrow asset
    MockERC20 wbtc;   // allowed collateral 1
    MockERC20 wsteth; // allowed collateral 2
    MockERC20 eth;    // victim wants to use this as collateral but will be blocked

    EscrowLocal escrowWbtc;
    EscrowLocal escrowWsteth;
    EscrowLocal escrowEth;

    uint256 DOMAIN_ID;

    function setUp() public {
        DOMAIN_ID = block.chainid;

        usdc   = new MockERC20("USD Coin", "USDC");
        wbtc   = new MockERC20("Wrapped BTC",  "wBTC");
        wsteth = new MockERC20("Wrapped stETH", "wstETH");
        eth    = new MockERC20("Wrapped ETH",  "WETH");

        oracle = new ValueOracle(admin, admin);

        CreditHub impl = new CreditHub(IERC20(address(usdc)), 100_00, 50, 12_00);
        bytes memory initData = abi.encodeCall(
            CreditHub.initialize,
            (IOracle(address(oracle)), admin, admin, admin, admin, admin, admin, 100_00)
        );
        creditHub = CreditHub(address(new ERC1967Proxy(address(impl), initData)));

        // CreditHubController: deploy then grant it CONTROLLER_ROLE on the hub.
        creditHubController = new CreditHubController(address(creditHub), admin,
        ↪  DOMAIN_ID, admin);
        vm.prank(admin);
        creditHub.grantRole(bytes32("CONTROLLER_ROLE"), address(creditHubController));

        // Deploy EscrowLocals and register each collateral in CreditHub.
        escrowWbtc   = _deployAndRegister(wbtc);
        escrowWsteth = _deployAndRegister(wsteth);
        escrowEth    = _deployAndRegister(eth);
    }

    // Deploys an EscrowLocal for `token` and registers it as a valid collateral in
    ↪  CreditHub.
    function _deployAndRegister(MockERC20 token) internal returns (EscrowLocal escrow) {
        escrow = new EscrowLocal(IERC20(address(token)), token.name(), token.symbol());

        // originAssetId = bytes32(escrow address), matching
        ↪  CreditHubController._addressToBytes32
        bytes32 originAssetId = bytes32(uint256(uint160(address(escrow))));
        bytes32 globalAssetId = originAssetId; // reuse for simplicity

        vm.prank(admin);
        creditHub.registerCollateral(DOMAIN_ID, originAssetId, globalAssetId);
        vm.prank(admin);
        creditHub.setCollateralConfig(globalAssetId, 70_00, 80_00, 5_00);
    }

    function testPOC() public {
        // Attacker deposits 1 wei of wBTC and wstETH with receiver = victim,
        // consuming both of victim's allowed collateral slots (MAX_GLOBAL_ASSETS = 2).
        wbtc.mint(attacker, 1);
        wsteth.mint(attacker, 1);

        vm.startPrank(attacker);
        wbtc.approve(address(escrowWbtc), 1);
```

```
        escrowWbtc.depositAndLock(1, victim, address(creditHubController));

        wsteth.approve(address(escrowWsteth), 1);
        escrowWsteth.depositAndLock(1, victim, address(creditHubController));
        vm.stopPrank();

        // Victim tries to deposit ETH — reverts because the collateral limit is already
        ↪  reached.
        eth.mint(victim, 1 ether);
        vm.startPrank(victim);
        eth.approve(address(escrowEth), 1 ether);
        vm.expectRevert(CreditHubBase.TooManyGlobalAssets.selector);
        escrowEth.depositAndLock(1 ether, victim, address(creditHubController));
        vm.stopPrank();
    }
}
```

Recommendation: Do not automatically add an asset to `$.collaterals[receiver]` on deposit. Instead, require users to explicitly opt an asset in as collateral via a separate method (e.g. `enableCollateral(globalAssetId)`). This also aligns with how users are expected to set their `liquidationOrder` and eliminates the need for a separate `collaterals` array distinct from `origins`.

Sprinter Tech: Acknowledged, won't fix. User won't lose anything (except of some temper perhaps). In case of such grieving, user could withdraw unneeded collateral. If such behavior will start causing trouble, we will implement additional user control.

Cantina Managed: Acknowledged.

3.2.4 `previewRedeem` in `ChainlinkOracle` Reverts for ERC7540 Async Vaults

Severity: Medium Risk

Context: ChainlinkOracle.sol#L95-L98, PreprocessorBase.sol#L112-L115

Description: When valuing ERC4626 vault shares, `getAssetValue` calls `IERC4626(asset).previewRedeem(amount)`. Per the ERC4626 standard, `previewRedeem` is explicitly allowed to revert. If it does, `getAssetValue` reverts and every function that relies on oracle pricing becomes unavailable for that collateral type.

ERC7540 async vaults are a concrete case: `previewRedeem` MUST revert on them by spec, so they can never be supported as collateral with the current implementation. `PreprocessorBase` contains helpers for 7540 vault shares and that indicates protocol might want to use them as collateral.

Recommendation: Replace `previewRedeem` with `convertToAssets`. One thing to note is that withdrawal fees are not reflected in `convertToAssets`.

Sprinter Tech: Fixed in PR 28 by using `convertToAssets` in case of 7540.

Cantina Managed: Fix verified.

3.2.5 Liquidation outcome is non monotonic with respect to the requested amount

Severity: Medium Risk

Context: CreditHubBase.sol#L566-L574

Description: The function `liquidate` from contract `CreditHubBase` can reject a smaller liquidation amount while accepting a larger one for the same borrower position.

This behavior comes from the interaction between the upward rounding used when converting the repayment budget into seized collateral and the final post liquidation check that requires the resulting position to either improve the maintenance health factor or end in a state where `totalCollateralAfter < debtAfter`.

As a result, liquidation success is not monotonic in the requested repayment amount. A partial liquidation can revert with `HealthFactorTooLow` because the rounded collateral seizure makes the

maintenance health factor slightly worse while the position is still not yet in bad debt. A larger liquidation against the same borrower can succeed because it pushes the account far enough that the `totalCollateralAfter < debtAfter` branch becomes true.

This creates an execution hazard for liquidators and integrations that probe with smaller partial amounts first. A failed smaller liquidation does not imply that the borrower is not liquidatable, since a larger repayment amount may still succeed immediately afterward.

A concrete reachable state is:

- Collateral deposited: `104.000000`.
- Borrow principal: `99.000000`.
- Fixed fee: `0.495000`.
- Total debt at liquidation: `99.495000`.
- Collateral price moved from `$2.00` at borrow time to `$1.00` at liquidation time.
- Resulting maintenance health factor: `9407`.

In that state:

- `liquidate(10.000000)` reverts with `HealthFactorTooLow`.
- `liquidate(91.000000)` succeeds and seizes `95.550000` collateral.

Numerically:

- After `10.000000`, the borrower is left with `93.500000` collateral and `89.495000` debt, so the position is still not in bad debt, but the maintenance health factor drops from `9407` to `9402`, and the call reverts.
- After `91.000000`, the borrower is left with `8.450000` collateral and `8.495000` debt, so `totalCollateralAfter < debtAfter` becomes true, and the call succeeds.

Proof Of Concept:

```
// SPDX-License-Identifier: LGPL-3.0-only
pragma solidity ^0.8.28;

import {Test} from "../node_modules/forge-std/src/Test.sol";
import {TransparentUpgradeableProxy} from
    "../node_modules/@openzeppelin/contracts/proxy/transparent/TransparentUpgradeablePro
    ↪  xy.sol";
import {ChainlinkOracle} from "../contracts/ChainlinkOracle.sol";
import {CreditHub} from "../contracts/CreditHub.sol";
import {CreditHubBase} from "../contracts/CreditHubBase.sol";
import {TestUSDC} from "../contracts/testing/TestUSDC.sol";

contract LiquidationNonMonotonicityPoC is Test {
    uint256 internal constant DOMAIN_ID = 4;
    bytes32 internal constant GLOBAL_ASSET_ID = bytes32("USD");
    bytes32 internal constant ORIGIN_ASSET_ID = bytes32(uint256(uint160(0x1234)));

    address internal admin = makeAddr("admin");
    address internal controller = makeAddr("controller");
    address internal registrator = makeAddr("registrator");
    address internal user = makeAddr("user");
    address internal liquidator = makeAddr("liquidator");
    address internal profitAdmin = makeAddr("profitAdmin");
    address internal pauser = makeAddr("pauser");

    TestUSDC internal asset;
    ChainlinkOracle internal oracle;
    CreditHub internal creditHub;

    function setUp() public {
        asset = new TestUSDC();
        oracle = new ChainlinkOracle(admin, admin);
```

```
        CreditHub implementation = new CreditHub(asset, 85_00, 50, 12_00);
        bytes memory initData = abi.encodeCall(
            CreditHub.initialize,
            (oracle, controller, registrator, admin, liquidator, profitAdmin, pauser,
            ↪  100_00)
        );

        TransparentUpgradeableProxy proxy =
            new TransparentUpgradeableProxy(address(implementation), address(this),
            ↪  initData);
        creditHub = CreditHub(address(proxy));

        asset.mint(address(creditHub), 1000_000000);

        vm.prank(admin);
        oracle.setStaticPrice(GLOBAL_ASSET_ID, 2_000000, 6,
        ↪  ChainlinkOracle.AssetType.RAW);

        vm.prank(registrator);
        creditHub.registerCollateral(DOMAIN_ID, ORIGIN_ASSET_ID, GLOBAL_ASSET_ID);

        vm.prank(registrator);
        creditHub.setCollateralConfig(GLOBAL_ASSET_ID, 90_00, 90_00, 5_00);
    }

    function test_poc_realisticValues_smallLiquidationRevertsWhileLargerSucceeds()
    ↪  public {
        vm.prank(controller);
        creditHub.deposit(user, DOMAIN_ID, ORIGIN_ASSET_ID, 104_000000);

        vm.prank(user);
        creditHub.drawCredit(user, user, 99_000000);

        vm.prank(admin);
        oracle.setStaticPrice(GLOBAL_ASSET_ID, 1_000000, 6,
        ↪  ChainlinkOracle.AssetType.RAW);

        (,,,,, uint256 mHealthFactorBefore) = creditHub.getUserData(user);
        assertEq(mHealthFactorBefore, 9407, "unexpected initial mHealthFactor");

        asset.mint(liquidator, 91_000000);
        vm.prank(liquidator);
        asset.approve(address(creditHub), 91_000000);

        vm.prank(liquidator);
        vm.expectRevert(CreditHubBase.HealthFactorTooLow.selector);
        creditHub.liquidate(user, 10_000000);

        vm.prank(liquidator);
        uint256 repaid = creditHub.liquidate(user, 91_000000);
        assertEq(repaid, 91_000000, "unexpected repaid amount");

        (, uint256 debtAfter,, uint256 totalCollateralAfter,, uint256
        ↪  mHealthFactorAfter) = creditHub.getUserData(user);
        assertEq(totalCollateralAfter, 8_450000, "unexpected collateral after");
        assertEq(debtAfter, 8_495000, "unexpected debt after");
        assertEq(mHealthFactorAfter, 8952, "unexpected post-liquidation mHealthFactor");
        assertEq(
            creditHub.liquidatedAmount(liquidator, user, DOMAIN_ID, ORIGIN_ASSET_ID),
            95_550000,
            "unexpected seized collateral"
        );
    }
}
```

Recommendation: Consider to make liquidation behavior monotonic with respect to the requested amount, so that smaller valid liquidation attempts do not fail while larger ones succeed for the same position.

Consider also reviewing the interaction between liquidation rounding and the post liquidation validation rule so that partial liquidations remain predictable for liquidators and integrations.

Sprinter Tech: Fixed in PR 28.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 Zero amount withdrawals can arbitrarily advance the global withdrawal nonce

Severity: Low Risk

Context: CreditHubBase.sol#L428-L450

Description: The function `withdraw` from contract `CreditHubBase` does not validate that `amount` is non-zero before decreasing collateral state, validating the health factor, and incrementing `withdrawalNonce`.

As a result, an authorized caller can execute zero amount withdrawals successfully. Even though these calls do not release collateral, they still emit a `Withdraw` event and consume one value from the global `withdrawalNonce` sequence.

This behavior allows unnecessary advancement of the nonce space and generation of meaningless withdrawal events. Since the same global nonce is also used by remote withdrawal and liquidation claim flows, this introduces avoidable noise in the cross domain settlement pipeline and makes monitoring and off chain processing less clean than necessary.

Recommendation: Consider to reject zero amount withdrawals explicitly at the beginning of `withdraw` so that the function only emits events and advances `withdrawalNonce` for meaningful state transitions.

Sprinter Tech: Fixed in PR 28 by restricting 0 amount.

Cantina Managed: Fix verified.

3.3.2 Revocation finalization and borrowing remain simultaneously valid at the revoke boundary

Severity: Low Risk

Context: ExclusiveOperator.sol#L49

Description: The function `finalizeRevoke` from contract `ExclusiveOperator` allows revocation to be finalized when `accessUntil <= block.timestamp`.

If `openCreditLine` uses the same boundary condition to allow access until `block.timestamp == accessUntil`, both actions remain valid at the exact expiry timestamp. At that boundary, one transaction can still open credit while another can finalize the revocation in the same block, depending on ordering.

This creates an avoidable race condition at the revocation boundary and makes the effective cutoff ambiguous. The operator access is not cleanly separated from the moment when revocation becomes effective.

Recommendation: Consider to make the boundary conditions asymmetric so that only one of the two actions is valid at the exact cutoff timestamp. This would make the revocation point unambiguous and avoid race dependent behavior at expiry.

Sprinter Tech: Fixed in PR 28 by allowing revocation strictly after the `accessUntil`.

Cantina Managed: Fix verified.

3.3.3 Zero liquidation bonus weakens the collateral initialization guard

Severity: Low Risk

Context: CreditHubBase.sol#L331

Description: The function `setCollateralConfig` from contract `CreditHubBase` uses `liquidationBonus == 0` as the condition to determine whether a collateral configuration has already been initialized.

This makes the initialization guard dependent on a business parameter that can also be a valid configuration value. If a collateral is configured with `liquidationBonus = 0`, the contract cannot distinguish between an uninitialized entry and an already initialized entry with a zero bonus. As a result, the same collateral configuration can be set again through the initialization path.

This weakens the intended separation between the initial registration flow and later configuration updates. In practice, it reduces the value of restricting `updateCollateralConfig` to the higher privileged role, since a configuration initialized with a zero liquidation bonus can be modified again through the path that is supposed to be one time only.

Recommendation: Consider to track collateral configuration initialization with an explicit boolean flag, or use a dedicated sentinel that cannot overlap with a valid business value. This would make the one time initialization check independent from `liquidationBonus`.

Sprinter Tech: Fixed in PR 28 by introducing `CollateralStatus` as means to determine registration.

Cantina Managed: Fix verified.

3.3.4 Oracle configuration cannot be symmetrically cleared across pricing modes

Severity: Low Risk

Context: ChainlinkOracle.sol#L84-L88

Description: The function `clearChainlinkFeed` from contract `ChainlinkOracle` allows a configured Chainlink feed to be removed, but the oracle does not provide an equivalent clearing path for assets configured through `setStaticPrice`.

This creates an asymmetric configuration model between the two supported pricing modes. An asset can be configured either with a static price or with a Chainlink feed, and those modes are mutually exclusive. However, only the Chainlink path has an explicit clearing function.

As a result, if an asset is configured with a static price and later needs to be removed or reset, there is no direct way to do so through the static price path. In practice, the configuration would need to be transitioned through `setChainlinkFeed` and then cleared with `clearChainlinkFeed`, which is not a clean or direct state transition. In addition, once an asset has been configured, the contract does not provide a fully consistent mechanism to return it to a neutral unconfigured state across both pricing modes.

This makes the oracle state machine incomplete and may complicate operational handling when asset pricing configuration needs to be changed or removed.

Recommendation: Consider to provide a clearing path for static price configurations as well, or refactor the oracle configuration flow so both pricing modes follow a symmetric and explicit lifecycle. This would make the state transitions clearer and allow assets to be returned to an unconfigured state without relying on indirect intermediate steps.

Sprinter Tech: Fixed in PR 28 by fully clearing configs when setting price/feed.

Cantina Managed: Fix verified.

3.3.5 ERC4626 asset identifier validation is deferred to runtime instead of configuration time

Severity: Low Risk

Context: ChainlinkOracle.sol#L119

Description: The function `getAssetValue` from contract `ChainlinkOracle` validates that `assetId` fits into an address sized value when the asset is marked as ERC4626.

This means the correctness of the ERC4626 asset identifier is not enforced when the asset is configured, but only later when the asset is queried. As a result, an invalid ERC4626 `assetId` can be stored successfully and remain in the configuration until a protocol flow attempts to read its value.

Because asset valuation is used by core flows such as borrowing, withdrawals, and liquidation, storing an invalid ERC4626 identifier can make those flows revert when the oracle is consulted. This turns a configuration mistake into a runtime failure affecting normal protocol operation.

Validating the identifier only at query time also adds a check that is repeated on every valuation call, even though the property is static once the asset is configured.

Recommendation: Consider to validate the ERC4626 `assetId` when the asset configuration is set if `isERC4626_` is true. This would prevent invalid identifiers from being stored and avoid runtime failures in protocol flows that depend on oracle valuation.

Sprinter Tech: Fixed in PR 28 by performing validation during setup.

Cantina Managed: Fix verified.

3.3.6 Static price configuration does not bound decimals and can break oracle valuation

Severity: Low Risk

Context: ChainlinkOracle.sol#L49-L61

Description: The function `setStaticPrice` from contract `ChainlinkOracle` stores `decimals_` without enforcing the same upper bound that is already applied in the Chainlink feed configuration path.

The same pattern exists in the static pricing flow of `ValueOracle`, where `setPrice` also accepts arbitrary decimals. Both oracle implementations later compute `10 ** decimals` during valuation. If `decimals` is set above the safe exponent range for `uint256`, the exponentiation overflows and the valuation call reverts.

As a result, a single incorrect static price configuration can make asset valuation unusable for the affected collateral. Since oracle valuation is used by core protocol flows such as borrow capacity checks, withdrawals, and liquidation, this can block normal protocol operation for that asset.

The issue is more visible in `ChainlinkOracle` because the Chainlink feed path already validates decimals during configuration, while the static price path does not apply the same constraint. In `ValueOracle`, the static pricing flow is also inconsistent because it allows a zero price to be configured directly.

Recommendation: Consider to enforce an upper bound on static price decimals at configuration time, consistent with the bound already used in the Chainlink feed path.

Consider also validating that the configured static price is non zero where required, so invalid static configurations cannot be stored and later break valuation dependent flows.

Sprinter Tech: Fixed in PR 28. Zero price is not allowed anymore, decimals upper bound is added.

Cantina Managed: Fix verified.

3.3.7 Liquidation order accepts origins with zero borrower deposit

Severity: Low Risk

Context: CreditHubBase.sol#L678-L685

Description: The function `setLiquidationOrder` from contract `CreditHubBase` allows a borrower to include any registered origin in their liquidation order, even if they currently have zero deposit in that origin.

This means the stored liquidation order is not restricted to the borrower's actual active collateral set. As a result, the liquidation order can contain entries that are irrelevant for the borrower position and cannot contribute any collateral during liquidation.

Allowing zero deposit entries in the liquidation order makes the configuration less accurate and introduces unnecessary processing of origins that do not belong to the borrower's current effective collateral position.

Recommendation: Consider to restrict `setLiquidationOrder` so that each origin in the provided order must correspond to a borrower origin with a nonzero deposited amount at the time the order is set.

Consider also pruning entries that reach zero balance, so the stored liquidation order remains aligned with the borrower's active collateral positions.

Sprinter Tech: Acknowledged, won't fix. Liquidation logic tolerates invalid entries in the user provided liquidation order, and it is up to the user to keep it orderly if they care about the actual order.

Cantina Managed: Acknowledged.

3.3.8 Registered collateral can become depositable before it is priceable

Severity: Low Risk

Context: CreditHubBase.sol#L391-L400

Description: The function `deposit` from contract `CreditHubBase` allows collateral to be credited as soon as the `(originDomainId, originAssetId)` pair is registered, but it does not verify that the corresponding `globalAssetId` can already be valued by the configured oracle.

As a result, an admin can complete collateral registration before completing a valid oracle configuration for that asset. Once a user deposits such collateral, health factor dependent flows start iterating over the user's held collateral set and unconditionally query the oracle for each global asset. If valuation for the newly registered asset reverts, the affected account can no longer execute any flow that depends on collateral valuation.

On `ChainlinkOracle`, this can happen when the asset has no valid pricing source configured yet, or when the configured source is not currently usable. In that case, a user can end up holding collateral that is accepted by the hub but cannot be processed by borrow, withdraw, liquidation, or health factor read paths.

This creates an onboarding sequencing risk where one incorrect admin step can make a registered collateral depositable before it is safely priceable, and then block core protocol behavior for accounts holding it.

Recommendation: Consider to enforce during collateral onboarding that any newly registered collateral is already priceable by the active oracle before deposits are allowed for it.

Consider also validating the oracle path at configuration time, so a collateral cannot transition into a depositable state unless `getAssetValue` succeeds for that asset.

Sprinter Tech: Fixed in PR 28 by querying collateral value at registration and at oracle update.

Cantina Managed: Fix verified.

3.3.9 Zero amount repay can inflate borrower fees through repeated credit line refresh

Severity: Low Risk

Context: CreditHubBase.sol#L484-L502

Description: The function `repay` from contract `CreditHubBase` allows calls with `amount = 0` and still refreshes and stores the borrower's credit line state through `getUpdatedCreditLine`.

During the refresh step, carryover and grace fee accrual is computed using rounding up semantics. Because the function persists the refreshed credit line even when no repayment occurs, a third party can repeatedly call `repay(borrower, 0)` across different timestamps and force the borrower's credit line to persist additional rounded fee units.

Since no tokens are transferred when `amount = 0`, this behavior allows an attacker to increase a borrower's recorded fee balance without providing any repayment. Over multiple calls, this can accumulate additional fee units compared to a passive borrower whose credit line is only refreshed during normal interactions.

The effect becomes relevant for liquidation logic because accounts with positive fee balances can become actionable even when their maintenance health factor remains above the liquidation threshold. As a result, the inflated fees can lead to additional collateral being seized during liquidation compared to an identical borrower whose credit line was not externally refreshed.

Recommendation: Consider rejecting zero amount repayments so that `repay` cannot be used to refresh and persist borrower credit lines without performing an actual repayment. This would prevent third parties from artificially inflating borrower fee balances through repeated zero value calls.

Sprinter Tech: Fixed in PR 28 by denying 0 amount repay.

Cantina Managed: Fix verified.

3.3.10 Pause mechanism not strongly defined

Severity: Low Risk

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

Description: The pause mechanism does not consistently restrict all state changing flows that affect protocol risk and user positions.

As a result, entering a paused state does not provide a clear or uniform guarantee about which operations are still allowed and which ones are blocked. Some actions that change collateral or debt related state remain available, while other sensitive actions are disabled. This creates an ambiguous emergency mode where the protocol is paused, but important risk affecting transitions can still occur.

This ambiguity is relevant from a risk management perspective. In an emergency state, the protocol should have a clearly defined policy for whether users are only allowed to improve their positions, whether risk reducing actions remain available, and whether risk increasing or collateral removing actions must be blocked. Without a consistently enforced model, the paused state may not behave as expected during incidents and can leave room for outcomes that do not match the intended operational response.

In particular, if liquidation or other protection mechanisms are restricted while some position changing flows remain open, unhealthy positions may evolve in unintended ways and risk handling becomes dependent on operational assumptions rather than a clearly enforced contract level policy.

Recommendation: Consider to define an explicit pause policy for the protocol and enforce it consistently across all flows that can affect collateral, debt, liquidation handling, and overall protocol risk.

If the intended design is to allow only risk reducing actions during pause, consider to encode that rule directly in the implementation. If some exceptions are intentional, consider to document them clearly and ensure the resulting paused state still provides a coherent emergency mode.

Sprinter Tech: Fixed in PR 28 by adding pausing information to the contract description. All the outflow must be restricted, inflow allowed. So only deposit/repay can still be done during pause.

Cantina Managed: Fix verified.

3.3.11 `maxWithdraw` can overestimate removable collateral for origins that share a global asset

Severity: Low Risk

Context: CreditHubBase.sol#L819-L838

Description: The function `maxWithdraw` from contract `CreditHubBase` computes the removable amount for a specific origin using that origin in isolation, while the borrower credit capacity is accounted on the aggregated collateral amount of the corresponding `globalAssetId`.

When multiple origins map to the same `globalAssetId`, this creates a mismatch in the rounding path. `getUserData` derives borrowing capacity from the total collateral aggregated at the global asset level, while `maxWithdraw` values only the selected origin and applies the LTV to that origin specific amount. Because both valuation and LTV application round down, the isolated origin calculation is not equivalent to recomputing the borrower position after removing that origin amount from the aggregated global collateral.

As a result, `maxWithdraw` can return an amount that is not actually withdrawable. A borrower can receive a positive result from `maxWithdraw` for one origin, but attempting to withdraw that exact amount can still revert with `HealthFactorTooLow` once the protocol evaluates the true post withdrawal aggregate position.

This makes the view result unreliable for integrators and users when the same global asset is backed by multiple origins and the position is close to its LTV boundary.

Recommendation: Consider to compute `maxWithdraw` from the borrower post withdrawal aggregate collateral amount for the relevant `globalAssetId`, using the same aggregation and rounding path as `getUserData`.

This would keep the view result aligned with the actual health factor validation enforced by `withdraw`.

Sprinter Tech: Fixed in PR 28 by overestimating the selected asset credit capacity and underestimating the total select collateral credit capacity (same as in HF calc). Take the less out of two.

Cantina Managed: Fix verified.

3.3.12 `EscrowVault` Uses Default `decimalOffset` of Zero, Leaving Inflation Attack Possible

Severity: Low Risk

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

Description: `EscrowVault` inherits from OpenZeppelin's ERC4626 but does not override `_decimalsOffset()`, leaving it at the default of 0. With a zero offset, a classic inflation attack (donate large amounts to manipulate the share price before the first depositor) is theoretically possible, potentially causing early depositors to lose part of their deposit to the attacker's inflated shares. The attack is not profitable with offset 0 because the value captured by virtual shares from the attacker's donation matches the attacker's expected gains but it can be made to harm first few depositors.

Recommendation: Override `_decimalsOffset()` to return a larger value). A higher offset makes the attack exponentially more expensive than it is profitable.

Sprinter Tech: Acknowledged, won't fix. As the OZ documentation says even with default 0 value it is unprofitable for the attacker.

Cantina Managed: Acknowledged.

3.3.13 Suggestion on How to Protect `repayWithPermit` from Griefing

Severity: Low Risk

Context: CreditHubBase.sol#L526-L548

Description: There is already an acknowledgement of this attack in a comment that a malicious actor could frontrun the `permit()` call in separate transaction, which will make `repayWithPermit()` revert. While the attacker gains nothing from this, it degrades the UX and forces users to retry with directly calling the repay function.

Recommendation: Wrap the `permit()` call in a `try/catch`. If it reverts (because the permit signature was already used by the frontrunner), skip it and proceed to repay:

```
function repayWithPermit(
    address borrower,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external returns(uint256, uint256) {
    try IERC20Permit(address(ASSET)).permit(
        _msgSender(),
        address(this),
        amount,
        deadline,
        v,
        r,
        s
    ){} catch {}
    return repay(borrower, amount);
}
```

Sprinter Tech: Fixed in PR 28 through the suggested solution.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Add `EscrowHelper` usage documentation

Severity: Informational

Context: EscrowHelper.sol#L22-L24

Description: The function `depositAndLockAll` from contract `EscrowHelper` deposits and locks the entire token balance currently held by the helper contract.

Instead of receiving an explicit amount from the caller, the function reads `token.balanceOf(address(this))`, approves the escrow for that full balance, and forwards all tokens to `depositAndLock`.

This behavior assumes that the helper is used in an atomic execution flow where tokens are transferred to the helper and the helper function is executed in the same transaction, for example through a multicall integration. If tokens are transferred to the helper contract in a separate transaction and remain there temporarily, any later caller can invoke the helper and route the entire helper held balance to an arbitrary `receiver` and `controller`.

This does not break protocol accounting, but it introduces an integration risk if the expected usage pattern is not clearly documented.

Recommendation: Consider to document that `EscrowHelper` is intended to be used only in atomic execution flows where token transfers and helper execution occur within the same transaction, and that tokens should not remain stored in the helper contract between transactions.

Sprinter Tech: Fixed in PR 28 by extending the description and adding warnings.

Cantina Managed: Fix verified.

3.4.2 Use of assert on production code

Severity: Informational

Context: CreditHubController.sol#L126

Description: The function `_bytes32ToAddress` from contract `CreditHubController` uses `assert(uint256(id) <= type(uint160).max)` to validate that a `bytes32` value can be safely cast to an address.

This check validates user supplied identifiers such as `originAssetId` and `originReceiver` before converting them to an address. However, `assert` is intended for internal invariants that should never fail during normal execution. When validating external or user controlled input, `require` is more appropriate because it signals expected validation failures and avoids triggering panic errors.

Using `assert` in this context may cause the transaction to revert with a panic error if the condition fails, which is not aligned with typical input validation patterns.

Recommendation: Consider replacing the `assert` with a `require` statement that reverts with an explicit error when the identifier does not fit within the address range. This would align the validation with standard input checking practices.

Sprinter Tech: Fixed in PR 28 by replacing `assert` with `require`.

Cantina Managed: Fix verified.

3.4.3 Signed deposit path accepts local origin tuples

Severity: Informational

Context: CreditHubController.sol#L52-L70

Description: The function `deposit` from contract `CreditHubController` accepts signed deposit messages for any `(originDomainId, originAssetId)` pair, including tuples that correspond to the local domain.

In the intended design, local deposits are performed through the `depositLocal` path, which is triggered atomically from `EscrowLocal._lock`. This path ensures that the hub collateral accounting is synchronized with escrow shares minted and locked on the same chain.

Allowing the signed `deposit` path to accept tuples from the local domain makes the separation between the two flows less explicit. Although the replay protection on

`(originDomainId, originAssetId, nonce)` prevents reuse of legitimate local lock events, the signed deposit path is primarily intended for cross domain collateral crediting.

Restricting the signed deposit flow to remote origins would make the intended architecture clearer and reduce the risk of misuse of the remote deposit path for local tuples.

Recommendation: Consider rejecting signed deposits where `originDomainId` equals the local domain identifier and require local collateral to be credited exclusively through the `depositLocal` path. This would make the separation between local and cross domain deposit flows explicit.

Sprinter Tech: Fixed in PR 28 by restricting local domain in `deposit()`.

Cantina Managed: Fix verified.

3.4.4 Escrow design lacks a recovery path for controller lock failures

Severity: Informational

Context: Escrow.sol#L55

Description: The escrow locking model does not provide a recovery mechanism in case the configured controller becomes invalid or stops functioning correctly.

As a result, assets locked through the controller based flow may remain unavailable even when the issue is operational rather than caused by the user position itself. Since the lock is enforced at the escrow level and normal release depends on controller mediated flows, a controller malfunction can leave users unable to unlock or withdraw their collateral through the intended path.

This introduces an operational recovery gap in the escrow design. In practice, if the controller becomes misconfigured, deprecated, or otherwise unusable, the protocol may have no direct contract level mechanism to restore access to affected locked positions.

Recommendation: Consider to introduce a carefully scoped recovery mechanism that can unlock positions when the controller is no longer valid or functional.

If such a mechanism is added, consider to restrict it through appropriate governance or admin controls and define clear conditions for its use so that it serves only as an emergency recovery path.

Sprinter Tech: Acknowledged, won't fix. Added warnings.

Cantina Managed: Acknowledged.

3.4.5 Cross chain deposit signing depends on offchain finality handling

Severity: Informational

Context: CreditHubController.sol#L52-L83

Description: The function `deposit` from contract `CreditHubController` credits hub collateral based on an offchain signature over a lock event payload from another domain.

This design assumes that the signing process only authorizes finalized origin chain lock events. If a signature is produced before the origin chain reaches sufficient finality, a later reorg can invalidate the underlying lock event while the hub side collateral has already been credited on the destination chain.

As a result, the protocol can temporarily or permanently record collateral that is no longer backed by a valid origin chain lock. In a multichain design, this creates a dependency on the offchain signing process to enforce chain specific finality before authorizing deposits.

This is an expected trust assumption of the system, but it is operationally important because an incorrect signing policy can affect collateral correctness across chains.

Recommendation: Consider to document explicitly that cross chain deposit signatures must only be produced after sufficient finality on the origin domain.

Consider also defining chain specific finality requirements for the signing infrastructure so the offchain process applies a consistent and conservative confirmation policy before authorizing hub side deposits.

Sprinter Tech: Fixed in PR 28 by adding warnings to the controller descriptions.

Cantina Managed: Fix verified.

3.4.6 Admin liquidity withdrawal can desynchronize internal accounting invariants

Severity: Informational

Context: CreditHubBase.sol#L385-L390

Description: The function `withdrawLiquidity` from contract `CreditHubBase` transfers the full hub asset balance to an arbitrary receiver, but it does not update any internal accounting variables that derive protocol liquidity state.

As a result, the contract balance can be reduced independently from accounting values such as remaining liquidity and profit. This makes balance based reality diverge from the internal state that other parts of the system use to reason about available liquidity and protocol funds.

Given the intended use, this appears to be an operational or contingency function for migrations or liquidity management rather than a normal user flow. However, from an invariant perspective, it bypasses the accounting model and can leave the system in a state where internal liquidity related variables no longer reflect the actual asset balance held by the contract.

This is particularly relevant for protocol owned amounts such as accumulated profit, since the function can withdraw them without resetting the corresponding accounting state. In that situation, later logic may continue to treat profit as still present even though the assets have already been removed.

Recommendation: Consider to either update the relevant accounting variables when this function is used, or constrain the function so it can only withdraw the portion of funds that is intentionally meant to remain outside the normal accounting model.

If this function is intended only for contingency operations or migrations, consider to document that it bypasses standard liquidity accounting and ensure any related state, including profit if applicable, is reconciled as part of the same procedure.

Sprinter Tech: Fixed in PR 28 by removing the function.

Cantina Managed: Fix verified.

3.4.7 Unconfigured domain receivers can leave liquidation claims pending in the helper queue

Severity: Informational

Context: Liquidator.sol#L105-L108

Description: The function `claimAndPayback` from contract `Liquidator` skips claim execution when the receiver for an origin domain is unset and only emits `MissingReceiver`.

In that situation, the corresponding liquidated origin remains pending inside the helper state because removal only happens when the claim path reaches `HUB.claimLiquidatedDeposit` through `CREDIT_HUB_CONTROLLER.claim`. Since the skipped entry is not removed, repeated liquidations can continue adding pending claim entries for domains that do not have a configured receiver.

As a result, the helper queue can grow over time for those domains, and each later execution of `claimAndPayback` must iterate over a larger pending set. This does not immediately break the liquidation mechanism, especially since the missing receiver condition is expected to be operational and temporary, but it does create a liveness and maintainability concern for the helper path if the configuration gap persists.

Given the intended operating model, this appears to be an administrative contingency issue rather than a direct protocol vulnerability. Still, the current behavior allows unresolved claim entries to accumulate until a receiver is configured or the helper logic is upgraded.

Recommendation: Consider to document clearly that receivers must be configured for every supported origin domain before liquidations involving that domain are expected to be processed through the helper.

Consider also adding an operational recovery mechanism, such as paged claiming or another bounded processing approach, so pending entries can be handled safely if receiver configuration is delayed.

Sprinter Tech: Acknowledged. Comments added. In case of too long queue, we will introduce paging with an upgrade.

Cantina Managed: Acknowledged.

3.4.8 Liquidator helper cannot handle empty claim sets and reverts instead

Severity: Informational

Context: Liquidator.sol#L96-L100

Description: The function `claimAndPayback` from contract `Liquidator` starts by reading the number of pending liquidated origins and immediately enters a `do while` loop that decrements the index.

When the pending set is empty, the first `i--` underflows and the function reverts. This means the helper does not handle the zero pending claims case explicitly and instead relies on arithmetic underflow to abort execution.

In practice, this makes the helper behavior less clear and less robust from an operational perspective. If liquidation execution reaches a state where no claimable origins were added for the helper, `claimAndPayback` cannot complete gracefully and the refund path is not reached.

Based on the developer comment, this appears to be intentional behavior rather than an unintended security issue, since the expected policy is that a liquidation producing no claimable origins should revert. However, the current implementation expresses that policy indirectly through loop underflow instead of an explicit condition and error.

Recommendation: Consider to replace the current loop with an explicit empty set check and revert with a dedicated error when there are no pending liquidated origins to process.

Consider also keeping the helper control flow explicit so the intended behavior is clear and does not depend on arithmetic underflow.

Sprinter Tech: Fixed in PR 28 by explicitly reverting in case of empty liquidated origins.

Cantina Managed: Fix verified.

3.4.9 Missing Events on State Changing Functions

Severity: Informational

Context: EscrowController.sol#L41-L44, ExclusiveOperator.sol#L41-L44, Operator.sol#L44-L51

Description: Several state changing functions do not emit events, making off-chain monitoring and indexing harder.

- `Operator.addCreditReceiver` and `removeCreditReceiver`: no events emitted when receivers are added or removed.
- `EscrowController` constructor and `setSigner` function: sets `signer` but does not emit a `SignerSet(signer_)` event.
- `ExclusiveOperator`: sets `revokeScheduledAt[msg.sender]` but emits no event.

Recommendation: Emit appropriate events in each case.

Sprinter Tech: Fixed in PR 28 by adding events.

Cantina Managed: Fix verified.

3.4.10 Unused Custom Errors in `CreditHubBase`

Severity: Informational

Context: CreditHubBase.sol#L141, CreditHubBase.sol#L144-L145

Description: These custom errors are declared but never referenced anywhere in the codebase:

- `DepositAlreadyProcessed`.
- `CreditLineAlreadyOpened`.
- `InsufficientCollateral`.

Recommendation: Remove them to keep the interface clean.

Sprinter Tech: Fixed in PR 28 by removing unused errors.

Cantina Managed: Fix verified.

3.4.11 `liquidationBonus >= 0` Check Is Always True

Severity: Informational

Context: CreditHubBase.sol#L315

Description:

```
require(liquidationBonus >= 0, InvalidCollateralConfig());
```

`liquidationBonus` is typed as a signed integer but the check `>= 0` is a tautology, it can never fail.

Recommendation: If a zero liquidation bonus is a valid config, remove the check entirely. If a positive bonus should be required, change the condition to `liquidationBonus > 0`.

Sprinter Tech: Fixed in PR 28 by removing the guard.

Cantina Managed: Fix verified.

3.4.12 `onlyOperatorOrBorrower` Modifier Name Is Misleading

Severity: Informational

Context: CreditHubBase.sol#L363-L370

Description: The modifier name `onlyOperatorOrBorrower` implies either the operator or the borrower can call the function unconditionally. In reality, the borrower can only call it if no operator is set.

Recommendation: Rename to `onlyOperatorIfSetOrBorrower` to accurately reflect the conditional logic.

Sprinter Tech: Fixed in PR 28 by renaming the modifier.

Cantina Managed: Fix verified.

3.4.13 Missing `debtors` Array Length View Function

Severity: Informational

Context: CreditHubBase.sol#L840-L842

Description: `getDebtorsForLiquidation` accepts `offset` and `limit` parameters to paginate through the debtors array, but there is no way for a caller to know the total array length without calling the function with an out-of-bounds offset and observing an empty result.

Recommendation: Expose a simple view function to make the helper function better:

```
function getDebtorsCount() external view returns (uint256) {
    return _getCreditHubStorage().debtors.length();
}
```

Sprinter Tech: Fixed in PR 28 by adding `getDebtorsLength()` function.

Cantina Managed: Fix verified.

3.4.14 `drawCredit` transfer credit asset before enforcing the hub invariants

Severity: Informational

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

Description: `drawCredit` now transfers the credit asset before enforcing the hub invariants. That creates a reentrancy window for non standard callback assets. In CreditHubBase.sol:484, debt is recorded and `CREDIT_ASSET.safeTransfer(receiver, amount)` runs before `_validateHealthFactor`, `_validateUtilization`, and `_validatePerBlockBorrowLimit`. A receiver can tehen repay inside the transfer hook and make those checks observe the post repay state instead of the borrowed state.

With a hook token it is possible that a borrower with zero collateral successfully borrows and repays 1000 in the transfer callback, and `drawCredit` still completed.

Note that this requires `CREDIT_ASSET` to have a callback non standard functionality. This can be understood as a non intended flash loan functionality.

Recommendation: Consider doing using CEI pattern.

Sprinter: Acknowledged, won't fix. As mentioned it needs a token that passes execution under receivers control, and we don't have plans to use such credit assets. Besides if everything checks out in the system in the end, then it is acceptable.

Sprinter Tech: Acknowledged, won't fix. As you've mentioned it needs a token that passes execution under receivers control, and we don't have plans to use such credit assets. Besides if everything checks out in the system in the end, then it is acceptable.

Cantina Managed: Acknowledged.

3.4.15 Unbounded oracle rotation path

Severity: Informational

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

Description: The new oracle rotation path is unbounded and can become unusable as registrations grow. `CreditHubBase.sol:288` loops over every element of `registeredCollateralIds`, while `CreditHubBase.sol:324` with no cap or cleanup path.

A large enough registry, whether organic or caused by a bad `REGISTRATOR_ROLE` holder, can make emergency oracle replacement exceed practical gas limits.

Recommendation: Consider the next flow to avoid the previous described scenario:

1. Make oracle rotation O(1).
2. Move validation into a separate paginated flow.
3. Optionally add a cleanup or global cap if the protocol intends a small asset set.

This can preserve on chain validation without making emergency replacement uncallable.

Sprinter Tech: Acknowledged, won't fix. Added comments. If such thing ever happen, we would likely remove set validation.

Cantina Managed: Acknowledged.

3.4.16 Liquidation math does not account for collateral unit granularity and can break partial liquidation behavior

Severity: Informational

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

Description: The function `_liquidateOrigin` from contract `CreditHubBase` computes the seized collateral amount using upward rounding in token units, while the reward accounting is based on value calculations.

This approach assumes that collateral has sufficient unit granularity so that rounding effects remain negligible. However, when collateral has coarse units or when positions are small, rounding to whole token units can introduce a mismatch between the intended reward budget and the actual seized amount.

In such cases:

- Partial liquidations can become mathematically impossible because the rounded seizure would violate the post liquidation validation.
- Smaller liquidation attempts can revert while only large or full liquidations succeed.
- The seized collateral can deviate from the intended reward budget by up to one unit of the collateral.

The effect exists in general due to the interaction between unit based rounding of `liquidateAmount` and value based accounting of the reward. However, its practical impact depends on the granularity of the collateral:

- For coarse assets, the deviation can be large and materially affect liquidation behavior.
- For high precision assets, the deviation is typically limited to a small value and may be negligible.

There is no constraint in the collateral onboarding or oracle configuration that enforces a minimum unit precision or prevents listing assets where this mismatch becomes significant.

An example on a 6 decimal asset:

- Collateral config: `ltv = 90_00`, `mLtv = 90_00`, `liquidationBonus = 5_00`.
- Collateral amount: 2.000000.
- Collateral price at borrow: $100.000000.
- Collateral price at liquidation: $99.000000.
- Borrow principal: 178.000000.
- Fixed fee: 0.890000.
- Debt at liquidation: 178.890000.

The values obtained from this case are:

- Ideal reward value: 187.834500.
- Actual seized amount: 1.897318.
- Actual seized value: 187.834482.
- Shortfall: 0.000018.

So the bug still exists for a 6 decimal asset, but in this configuration it is dust-sized, not the large loss making case from the 0 decimal non realistic case.

Recommendation: Consider to account explicitly for collateral unit granularity in the liquidation logic, for example by aligning rounding between value and amount calculations or by bounding the deviation introduced by rounding.

Consider also enforcing constraints during collateral onboarding to ensure that listed assets have sufficient unit precision for the liquidation math to behave as intended.

If supporting coarse or low precision assets is required, consider documenting the limitations and ensuring liquidation behavior remains predictable under those conditions.

Sprinter Tech: Acknowledged, behavior is intended by design.

Added a test with explanation in commit 27beea6a.

User with a single unit of a very expensive collateral might not be liquidated until their debt grows enough so that liquidation bonus gets over the value of their single unit. This is fine.

Cantina Managed: Acknowledged.