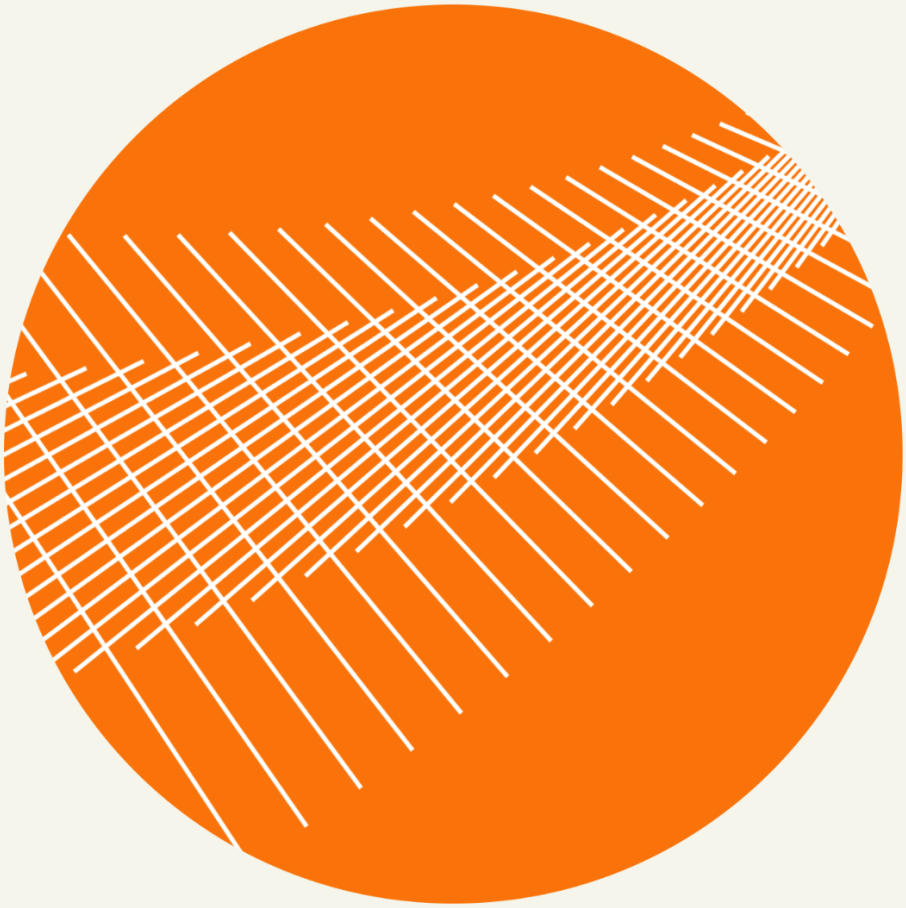
CTDSEC

YOU ARE PROTECTED



TERMS

Smart contract security audit

URM CDP - PASS

Table of Contents

1.0 Introduction

1.1 Project engagement

During July of 2026, URM CDP team engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. URM CDP provided CTDSec with access to their code repository and whitepaper.

1.2 Disclaimer

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the network's fast-paced and rapidly changing environment, we at CTDSec recommend that URM CDP team put in place a bug bounty program to encourage further and active analysis of the smart contract.

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the URM CDP contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

Source file:

contracts.zip [SHA256]: 3ff31492556d1c117da443971316f467fa06841ca597a860bc49996b4b93a624

Fix commit: 7d216b1682f7f7a51ee67bdd0c1fc5eec739e74e

2.2 Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

№	Issue description.
1	Compiler warnings.
2	Race conditions and Reentrancy. Cross-function race conditions.
3	Possible delays in data delivery.
4	Oracle calls.
5	Front running.
6	Timestamp dependence.
7	Integer Overflow and Underflow.
8	DoS with Revert.
9	DoS with block gas limit.
10	Methods execution permissions.
11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.
12	The impact of the exchange rate on the logic.
13	Private user data leaks.
14	Malicious Event log.
15	Scoping and Declarations.
16	Uninitialized storage pointers.
17	Arithmetic accuracy.
18	Design Logic.

19	Cross-function race conditions.
20	Safe Zeppelin module.
21	Fallback function security.
22	Overpowered functions / Owner privileges

3.0 Security Issues

3.1 High severity issues [3]

1. Production Adapters Hard-Code Dead CDP and NFT Addresses [Acknowledge ✅]

Location:

contracts/UrmCdpToken.sol:21-22

contracts/UrmCdp.sol:215

Issue: Production collateral adapters inherit URM_CDP and URM_CDP_NFT set to dead address. Adapter registration therefore fails because overview() reports a CDP address different from the live UrmCdp, all adapter user flows would also call dead addresses. The production CDP cannot operate.

Recommendation: Pass the live CDP and NFT addresses through the adapter constructor and store them as immutables. Validate them during deployment and integration testing.

2. Solvency and Liquidation Ignore Live URM Appreciation [Solved by design ✅]

Location:

contracts/UrmCdp.sol:331-333

contracts/UrmCdp.sol:493-497

contracts/UrmCdp.sol:565-571

contracts/UrmCdp.sol:621-627

Issue: Borrow sizing assumes URM is worth $1, while debt valuation later uses the live URM oracle price. Auto-unwind and emergency close-out only depend on the collateral-token price. If URM appreciates, a position can become economically underwater without becoming liquidatable, creating bad-debt exposure.

Recommendation: Use live URM price consistently in borrow sizing and liquidation logic, or add an LTV/debt-value liquidation trigger independent of collateral-price movement.

Fix: CDP consistently treats URM debt as $1-pegged, removing mixed live/fixed pricing. We recommend to verify Fortress and Flanking Tower provide a permissionless, uncapped, continuously funded $1.01 sell wall. Otherwise use a conservative live URM risk price for liquidation.

3. Permissionless Minimum Borrows Can Compound Victim Debt and Force Premature Liquidation [Solved ✅]

Location:

contracts/UrmCdp.sol:443

contracts/UrmCdp.sol:710

Issue: Any user can repeatedly open, wait, settle, and reopen a 1-URM position. Each open checkpoints the linear borrow index, compounding interest for all borrowers. At 98% utilization, this makes a victim liquidatable after 109 daily cycles while an identical control remains healthy.

Recommendation: Make interest accrual checkpoint-frequency independent, do not fold linear interval growth into the next interval's principal.

Fix: openPosition no longer calls repriceAccrual, so minimum borrows cannot capitalize victim debt.

3.2 Medium severity issues [5]

1. Auto-Unwind Buffer Validation Is Mathematically Insufficient [Solved ✅]

Location:

contracts/UrmCdp.sol:140-145

contracts/UrmCdp.sol:566-567

contracts/UrmCdp.sol:606-609

Issue: The protocol accepts autoUnwindTriggerBuffer >= autoUnwindFee + slippage, but this additive rule does not cover slippage applied to debt plus fee. An accepted configuration can therefore reach the liquidation trigger while still being unable to acquire enough URM to repay debt, reverting auto-unwind.

Recommendation: Enforce a multiplicative solvency condition, such as buffer >= 1 / (1 - slippage - fee) - 1, with rounding up and a safety margin. Reject configurations where fee plus slippage reaches 100%.

Fix: Validation now enforces the multiplicative fee-and-slippage solvency condition. Test against the real router with rounding, thin liquidity, price movement, and collateral-specific token behavior.

2. Debt Reductions Leave the Interest-Rate Checkpoint Stale [Solved ✅]

Location:

contracts/UrmCdp.sol:548-552

contracts/UrmCdp.sol:600-604

contracts/UrmCdp.sol:651-656

contracts/UrmCdp.sol:710-725

Issue: closePosition, autoUnwind, and closeOut reduce total debt but do not refresh the utilization-based interest checkpoint. Remaining borrowers can continue accruing at a previously extreme rate after utilization has fallen substantially, causing excess interest charges.

Recommendation: Settle the borrow index before changing debt, then call repriceAccrual() after every debt-reducing operation so subsequent accrual uses current utilization.

Comment: No state-transition repricing was added. After debt changes, accrue at the old rate, change debt, then checkpoint the rate at new utilization. A rpow-based index makes this safe.

Fix: closePosition and autoUnwind now reprice after reducing debt, so those paths correctly refresh the rate for remaining borrowers.repriceAccrual() was also added immediately after the closeOut debt/accounting decrements.

3. Oracle Unavailability Freezes Voluntary Repayment and Both Liquidation Paths [Solved ✅]

Location:

contracts/UrmCdp.sol:492

contracts/UrmCdp.sol:562

Issue: Settlement, auto-unwind, and close-out all require live oracle calls. A reverting oracle prevents a borrower with enough URM from repaying and blocks both forced-resolution paths.

Recommendation: Add an oracle-independent full-repayment path and bounded emergency liquidation handling using strictly freshness-limited fallback pricing.

Fix: Collateral-price failures are caught in the close overview, while URM repayment remains possible. Forced liquidation and close-out intentionally require the auto-unwind oracle. Document and operationally monitor this liveness tradeoff.

4. Different Allowed TWAP Windows Can Trigger an Unfundable Liquidation [Solved ✅]

Location:

contracts/UrmCdp.sol:98

contracts/UrmCdp.sol:565

Issue: The liquidation trigger uses autoUnwindTwap, while clearing-fee valuation uses CONFIG.twap. A valid short-window trigger combined with a divergent long-window valuation can require more URM than the full collateral can buy, reverting exact-output liquidation.

Recommendation: Use a consistent price basis for trigger, fee, and swap feasibility, or validate the worst-case permitted TWAP divergence.

Fix: Trigger, fee, collateral value, auto-unwind, and close-out use the auto-unwind TWAP. The global TWAP is informational only. autoUnwindTwap remains mutable for existing positions, snapshot it per position if stable borrower terms are required.

5. Valid Config Updates Can Strand Existing Positions Behind Stale Triggers [Solved ✅]

Location:

contracts/UrmCdp.sol:111

contracts/UrmCdp.sol:341

contracts/UrmCdp.sol:565

Issue: Positions snapshot their trigger at opening but use current fees, TWAP, and related liquidation settings. A configuration safe for new positions can make an old position trigger only after its collateral can no longer repay the newly calculated liquidation debt.

Recommendation: Snapshot all liquidation-critical parameters per position, or conservatively migrate active-position triggers during risk-increasing config updates.

Fix: Positions snapshot both fee and trigger buffer, leverage updates retain the stored buffer. Snapshot autoUnwindTwap for complete position-term isolation. Rebuild integrations because position structs and autoUnwindOverview ABI changed.

3.3 Low severity issues [0]

No low severity issues were found.

3.4 Informational Findings [0]

No informational severity issues were found

4.0 Testing coverage

During the testing phase, custom use cases were written to cover all the logic of contracts in python language. *Check “5 Annexes” to see the testing code.*

```
> urm-cdp-audit-v2@1.0.0 test
> hardhat test

Compiled 33 Solidity files successfully (evm target: paris).


  URM CDP audit harness
    √ functional: a correctly wired adapter requires both owners before registration (851ms)
    √ functional: opening and settling a borrow keeps CDP accounting synchronized (147ms)


  2 passing (999ms)
```

```
> urm-cdp-audit-v2@1.0.0 coverage
> hardhat coverage


Version
=======
> solidity-coverage: v0.8.17

Instrumenting for coverage...
============================

> AuditImports.sol
> Mocks.sol

Compilation:
============

Compiled 33 Solidity files successfully (evm target: paris).

Network Info
============
> HardhatEVM: v2.28.6
> network:    hardhat



  URM CDP audit harness
    ✓ functional: a correctly wired adapter requires both owners before registration (206ms)
    ✓ functional: opening and settling a borrow keeps CDP accounting synchronized (306ms)


  2 passing (516ms)

--------------------|----------|----------|----------|----------|----------------|
File                |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
--------------------|----------|----------|----------|----------|----------------|
 contracts_hardhat\ |    56.41 |       20 |    56.67 |    59.65 |                |
  AuditImports.sol  |      100 |      100 |      100 |      100 |                |
  Mocks.sol         |    56.41 |       20 |    56.67 |    59.65 |... 192,193,194 |
--------------------|----------|----------|----------|----------|----------------|
All files           |    56.41 |       20 |    56.67 |    59.65 |                |
--------------------|----------|----------|----------|----------|----------------|

> Istanbul reports written to ./coverage/ and ./coverage.json
```

5.0 Annexes

CDP Accounting Check:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "../../../contracts_under_audit/IERC20.sol";
import "../../../contracts_under_audit/IUrmCdp.sol";
import "../../../contracts_under_audit/IUrmOracle.sol";
import "../../../contracts_under_audit/UrmCdpStructs.sol";
import "../helpers/AuditMocks.sol";

contract MockSlippageCdpToken {
  uint256 internal constant BPS = 10000;

  IUrmCdp public immutable cdp;
  address public immutable collateralToken;
  uint256 public immutable swapSlippage;

  constructor(address cdp_, address collateralToken_, uint256
swapSlippage_) {
    cdp = IUrmCdp(cdp_);
    collateralToken = collateralToken_;
    swapSlippage = swapSlippage_;
  }

  function overview() external view returns
(UrmCdpStructs.UrmCdpTokenOverview memory) {
    return UrmCdpStructs.UrmCdpTokenOverview({
      symbol: "SLIP",
      tokenCa: collateralToken,
      cdp: address(cdp)
    });
  }

  function getPrice(uint256 quantity, uint32 twap) external view returns
(uint256) {
```

```
    return
IUrmOracle(AuditAddresses.URM_ORACLE).getRageTwapUsdcPrice(quantity, twap);
  }

  function tokenToCollateral(uint256 amount) external pure returns
(uint256) {
    return amount;
  }

  function collateralToToken(uint256 amount) external pure returns
(uint256) {
    return amount;
  }

  function openFor(address wallet, uint256 amount, uint8 collateralRatio)
external returns (uint256 nftId, uint256 urmReceived) {
    return cdp.openPosition(
      UrmCdpStructs.UrmCdpOpenPosition({
        cdpTokenAddr: address(this),
        wallet: wallet,
        tokenCollateral: amount,
        collateralRatio: collateralRatio,
        openTokenAmount: amount,
        openTokenType: 1,
        openType: 1
      })
    );
  }

  function closeOut(uint256 tokenAmount) external {
    require(msg.sender == address(cdp), "auth");
    require(IERC20(collateralToken).transfer(msg.sender, tokenAmount),
"xfer");
  }

  function autoUnwind(uint256 urmDebt, uint256 tokenCollateral, address,
uint256) external {
    require(msg.sender == address(cdp), "auth");
    uint256 tokenPrice =
IUrmOracle(AuditAddresses.URM_ORACLE).getRageTwapUsdcPrice(1e18, 60);
    uint256 urmPrice =
IUrmOracle(AuditAddresses.URM_ORACLE).getUrmTwapUsdcPrice(1e18, 60);
```

```
    uint256 denominator = tokenPrice * (BPS - swapSlippage);
    uint256 tokenNeeded = (urmDebt * urmPrice * BPS + denominator - 1) /
denominator;
    require(tokenNeeded <= tokenCollateral, "exact-out");

    require(IERC20(collateralToken).transfer(address(0xCAFE), tokenNeeded),
"sell");
    require(IERC20(AuditAddresses.URM).transfer(msg.sender, urmDebt),
"urm");
  }
}

contract CdpAccountingRecheckTest is AuditTestBase {
  function
test_H01_UrmAppreciationCreatesUndercollateralizedPositionWithoutLiquidatio
nTrigger() public {
    (UrmCdp cdp, MockGoodCdpToken adapter) = deployConfiguredHarness();
    MockERC20(AuditAddresses.RAGE).mint(address(adapter), 10_000e18);

    (uint256 nftId,) = adapter.openFor(AuditAddresses.USER, 10_000e18, 2);
    MockOracle(AuditAddresses.URM_ORACLE).setUrmPrice(2_200_000);

    UrmCdpStructs.UrmCdpClosePositionOverview memory closeView =
cdp.closePositionOverview(nftId);
    (, UrmCdpStructs.UrmCdpAutoUnwindOverview memory unwindView) =
cdp.autoUnwindOverview(nftId);

    require(closeView.debtValue > closeView.collateralValue, "not
undercollateralized");
    require(closeView.ltv > 10000, "ltv not underwater");
    require(!unwindView.canAutoUnwind, "unexpected liquidation trigger");

    vm.prank(AuditAddresses.OWNER1);
    vm.expectRevert(abi.encodeWithSignature("Error(string)", "can"));
    cdp.closeOut(nftId);
  }

  function
test_M01_BufferEqualToFeePlusSlippageDoesNotGuaranteeAutoUnwindSuccess()
public {
    installSystemMocks();
    UrmCdp cdp = new UrmCdp();
```

```
    MockERC20(AuditAddresses.URM).mint(address(cdp), 1_000_000e18);

    MockSlippageCdpToken adapter = new MockSlippageCdpToken(address(cdp),
AuditAddresses.RAGE, 2000);
    vm.prank(AuditAddresses.OWNER1);
    cdp.addToken(address(adapter), 1);
    vm.prank(AuditAddresses.OWNER2);
    cdp.addToken(address(adapter), 1);

    UrmCdpStructs.UrmCdpTokenConfig[9] memory configs;
    configs[0] = UrmCdpStructs.UrmCdpTokenConfig({
      borrowLimit: 250_000e18,
      minCollateralRatio: 2,
      maxCollateralRatio: 5,
      maxUrmDebt: 250_000e18,
      interestSurcharge: 0,
      slippage: 2000,
      autoUnwindTriggerBuffer: 3000,
      autoUnwindTwap: 900,
      autoUnwindFee: 1000
    });
    vm.prank(AuditAddresses.AUTOMATOR);
    cdp.setTokensConfigs(configs);

    UrmCdpStructs.UrmCdpToken memory token = cdp.getToken(address(adapter),
false);
    require(token.autoUnwindTriggerBuffer == token.autoUnwindFee +
token.slippage, "config not accepted at equality");

    MockERC20(AuditAddresses.RAGE).mint(address(adapter), 10_000e18);
    (uint256 nftId,) = adapter.openFor(AuditAddresses.USER, 10_000e18, 2);

    MockOracle(AuditAddresses.URM_ORACLE).setTokenPrice(650_000);
    (, UrmCdpStructs.UrmCdpAutoUnwindOverview memory unwindView) =
cdp.autoUnwindOverview(nftId);
    require(unwindView.canAutoUnwind, "not at trigger");

    vm.prank(AuditAddresses.AUTOMATOR);
    vm.expectRevert(abi.encodeWithSignature("Error(string)", "exact-out"));
    cdp.autoUnwind(nftId, 0);
  }
```

```
  function
test_M02_DebtReductionLeavesRemainingBorrowerAccruingAtStaleHighCheckpoint(
) public {
    (UrmCdp cdp, MockGoodCdpToken adapter) = deployConfiguredHarness();

    vm.warp(10_000);
    UrmCdpStructs.UrmCdpTokenConfig[9] memory configs;
    configs[0] = UrmCdpStructs.UrmCdpTokenConfig({
      borrowLimit: 500_000e18,
      minCollateralRatio: 2,
      maxCollateralRatio: 5,
      maxUrmDebt: 250_000e18,
      interestSurcharge: 0,
      slippage: 500,
      autoUnwindTriggerBuffer: 5000,
      autoUnwindTwap: 900,
      autoUnwindFee: 400
    });
    vm.prank(AuditAddresses.AUTOMATOR);
    cdp.setTokensConfigs(configs);

    MockERC20(AuditAddresses.RAGE).mint(address(adapter), 980_000e18);
    (uint256 firstNftId,) = adapter.openFor(AuditAddresses.USER,
490_000e18, 2);
    (uint256 secondNftId,) = adapter.openFor(AuditAddresses.USER,
490_000e18, 2);

    UrmCdpStructs.UrmCdpState memory peak = cdp.getState();
    require(peak.state.interestRateCheckpoint > 20000, "checkpoint not
high");

    vm.warp(20_000);
    UrmCdpStructs.UrmCdpClosePositionOverview memory firstClose =
cdp.closePositionOverview(firstNftId);
    MockERC20(AuditAddresses.URM).mint(address(adapter),
firstClose.urmDebtTotal);
    adapter.settleFromBalance(firstNftId, AuditAddresses.USER);

    UrmCdpStructs.UrmCdpState memory afterClose = cdp.getState();
    require(afterClose.utilization < 5000, "utilization did not fall");
    require(afterClose.currentInterestRate < 1500, "live rate did not
fall");
```

```
    require(afterClose.state.interestRateCheckpoint ==
peak.state.interestRateCheckpoint, "checkpoint refreshed");

    vm.warp(20_000 + 30 days);
    UrmCdpStructs.UrmCdpClosePositionOverview memory secondClose =
cdp.closePositionOverview(secondNftId);
    require(secondClose.urmInterest > 30_000e18, "remaining debt did not
accrue at stale high rate");
  }
}
```

Interest Checkpoint Frequency:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "../helpers/AuditMocks.sol";

/// @notice Regression for rate labels that promise APY while accrual
depends on
/// the number of Fortress checkpoints. Both positions below have identical
/// debt, utilization, price, and elapsed time.
contract M03_InterestCheckpointFrequencyTest is AuditTestBase {
  function _deployAtMaximumUtilization()
    internal
    returns (UrmCdp cdp, MockGoodCdpToken adapter, uint256 nftId)
  {
    cdp = new UrmCdp();
    MockERC20(AuditAddresses.URM).mint(address(cdp), 1_000_000e18);

    adapter = new MockGoodCdpToken(address(cdp), AuditAddresses.RAGE);
    vm.prank(AuditAddresses.OWNER1);
    cdp.addToken(address(adapter), 1);
    vm.prank(AuditAddresses.OWNER2);
    cdp.addToken(address(adapter), 1);

    // A one-day cadence is an explicitly permitted synchronization
interval.
    UrmCdpStructs.UrmCdpConfig memory config = cdp.getState().config;
    config.syncInterval = 1 days;
    vm.prank(AuditAddresses.AUTOMATOR);
```

```
    cdp.setConfigs(config);

    UrmCdpStructs.UrmCdpTokenConfig[9] memory configs;
    configs[0] = UrmCdpStructs.UrmCdpTokenConfig({
      borrowLimit: 500_000e18,
      minCollateralRatio: 2,
      maxCollateralRatio: 5,
      maxUrmDebt: 500_000e18,
      interestSurcharge: 0,
      slippage: 500,
      autoUnwindTriggerBuffer: 5000,
      autoUnwindTwap: 900,
      autoUnwindFee: 400
    });
    vm.prank(AuditAddresses.AUTOMATOR);
    cdp.setTokensConfigs(configs);

    // At the mock's $1 collateral and URM prices this opens exactly 500k
URM
    // against 1m collateral, placing the system at the default 250% APY
max.
    MockERC20(AuditAddresses.RAGE).mint(address(adapter), 1_000_000e18);
    (nftId,) = adapter.openFor(AuditAddresses.USER, 1_000_000e18, 2);

    UrmCdpStructs.UrmCdpState memory state = cdp.getState();
    require(state.utilization == 10_000, "not max utilization");
    require(state.state.interestRateCheckpoint == 25_000, "wrong rate");
  }

  function
test_M03_RoutineSyncFrequencyAloneChangesLiquidationEligibility() public {
    installSystemMocks();

    (UrmCdp checkpointed, , uint256 checkpointedId) =
_deployAtMaximumUtilization();
    (UrmCdp unsynced, , uint256 unsyncedId) =
_deployAtMaximumUtilization();

    // This is the configured cadence, rather than an adversarially
frequent
    // privileged call. The other CDP has exactly the same elapsed time and
    // utilization, but has not received a bookkeeping checkpoint.
```

```
    uint256 nextTimestamp = block.timestamp;
    for (uint256 checkpoint = 0; checkpoint < 43; checkpoint++) {
      nextTimestamp += 1 days;
      vm.warp(nextTimestamp);
      vm.prank(AuditAddresses.AUTOMATOR);
      checkpointed.fortressSync();
    }

    UrmCdpStructs.UrmCdpClosePositionOverview memory checkpointedClose =
checkpointed.closePositionOverview(checkpointedId);
    UrmCdpStructs.UrmCdpClosePositionOverview memory unsyncedClose =
unsynced.closePositionOverview(unsyncedId);
    (, UrmCdpStructs.UrmCdpAutoUnwindOverview memory checkpointedUnwind) =
checkpointed.autoUnwindOverview(checkpointedId);
    (, UrmCdpStructs.UrmCdpAutoUnwindOverview memory unsyncedUnwind) =
unsynced.autoUnwindOverview(unsyncedId);

    require(checkpointedClose.urmInterest > unsyncedClose.urmInterest +
15_000e18, "checkpoint impact too small");
    require(checkpointedClose.autoUnwindPrice > 1e6, "checkpointed trigger
below market");
    require(unsyncedClose.autoUnwindPrice < 1e6, "unsynced trigger above
market");
    require(checkpointedUnwind.canAutoUnwind, "checkpointed position not
liquidatable");
    require(!unsyncedUnwind.canAutoUnwind, "unsynced position unexpectedly
liquidatable");
  }

  function
test_M03_DailyCheckpointsTurnDefault250PercentApyIntoOver1100PercentInteres
t() public {
    installSystemMocks();
    (UrmCdp cdp, , uint256 nftId) = _deployAtMaximumUtilization();

    uint256 nextTimestamp = block.timestamp;
    for (uint256 checkpoint = 0; checkpoint < 365; checkpoint++) {
      nextTimestamp += 1 days;
      vm.warp(nextTimestamp);
      vm.prank(AuditAddresses.AUTOMATOR);
      cdp.fortressSync();
    }
```

```
    UrmCdpStructs.UrmCdpClosePositionOverview memory closeView =
cdp.closePositionOverview(nftId);
    // A 250% APY on a 500k position implies 1.25m annual interest. The
    // checkpointed linear-index formula charges more than 5m instead.
    require(closeView.urmInterest > 5_000_000e18, "interest did not exceed
1100 percent");
  }
}
```

Invariants:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "../helpers/AuditMocks.sol";

contract CDPInvariantHandler {
  Vm internal constant vm = Vm(address(uint160(uint256(keccak256("hevm
cheat code")))));

  UrmCdp internal immutable cdp;
  MockGoodCdpToken internal immutable adapter;
  uint256[] internal activeIds;

  constructor(UrmCdp cdp_, MockGoodCdpToken adapter_) {
    cdp = cdp_;
    adapter = adapter_;
  }

  function openSmallPosition(uint96 seedAmount) external {
    uint256 amount = 2_000e18 + (uint256(seedAmount) % 1_000e18);
    MockERC20(AuditAddresses.RAGE).mint(address(adapter), amount);
    try adapter.openFor(AuditAddresses.USER, amount, 2) returns (uint256
nftId, uint256) {
      activeIds.push(nftId);
    } catch {}
  }
```

```
  function settleNewestPosition() external {
    if (activeIds.length == 0) return;
    uint256 nftId = activeIds[activeIds.length - 1];
    activeIds.pop();

    vm.warp(block.timestamp + 1 hours + 1);
    try cdp.closePositionOverview(nftId) returns
(UrmCdpStructs.UrmCdpClosePositionOverview memory overview) {
      MockERC20(AuditAddresses.URM).mint(address(adapter),
overview.urmDebtTotal);
      try adapter.settleFromBalance(nftId, AuditAddresses.USER) {} catch {}
    } catch {}
  }
}

contract CDPAccountingInvariantTest is AuditTestBase {
  UrmCdp internal cdp;
  MockGoodCdpToken internal adapter;
  CDPInvariantHandler internal handler;
  address[] internal targets;

  function setUp() public {
    (cdp, adapter) = deployConfiguredHarness();
    handler = new CDPInvariantHandler(cdp, adapter);
    targets.push(address(handler));
  }

  // Foundry consumes this selector hook during invariant discovery.
Without it,
  // the handler actions are not guaranteed to be fuzzed.
  function targetContracts() external view returns (address[] memory) {
    return targets;
  }

  function invariant_TotalDebtAndCollateralMatchRegisteredTokenAccounting()
public {
    UrmCdpStructs.UrmCdpState memory state = cdp.getState();
    UrmCdpStructs.UrmCdpToken memory token = cdp.getToken(address(adapter),
false);

    require(state.state.totalUrmDebt == token.urmDebt, "debt mismatch");
    require(state.positionCount == token.positionCount, "position count
```

```
mismatch");
    require(MockERC20(AuditAddresses.RAGE).balanceOf(address(adapter)) >=
token.tokenCollateral, "collateral underfunded");
  }
}
```

6.0 Summary of the audit [PASS ✅]

Several high-medium severity gaps can disrupt user experience or disable features. With the recommended mitigations, the contract can achieve a robust security posture fit for production DeFi deployment. The audit result is failed until further review by the development team.

Following the development team's remediation review, all identified critical vulnerabilities have been addressed or appropriately assessed. Based on the evidence reviewed, the audit results are considered satisfactory.

Vulnerability Level	Total	Pending	Not Apply	Acknowledged	Partially Resolved	Resolved
High	3			1		2
Medium	5					5
Low						
Informational						