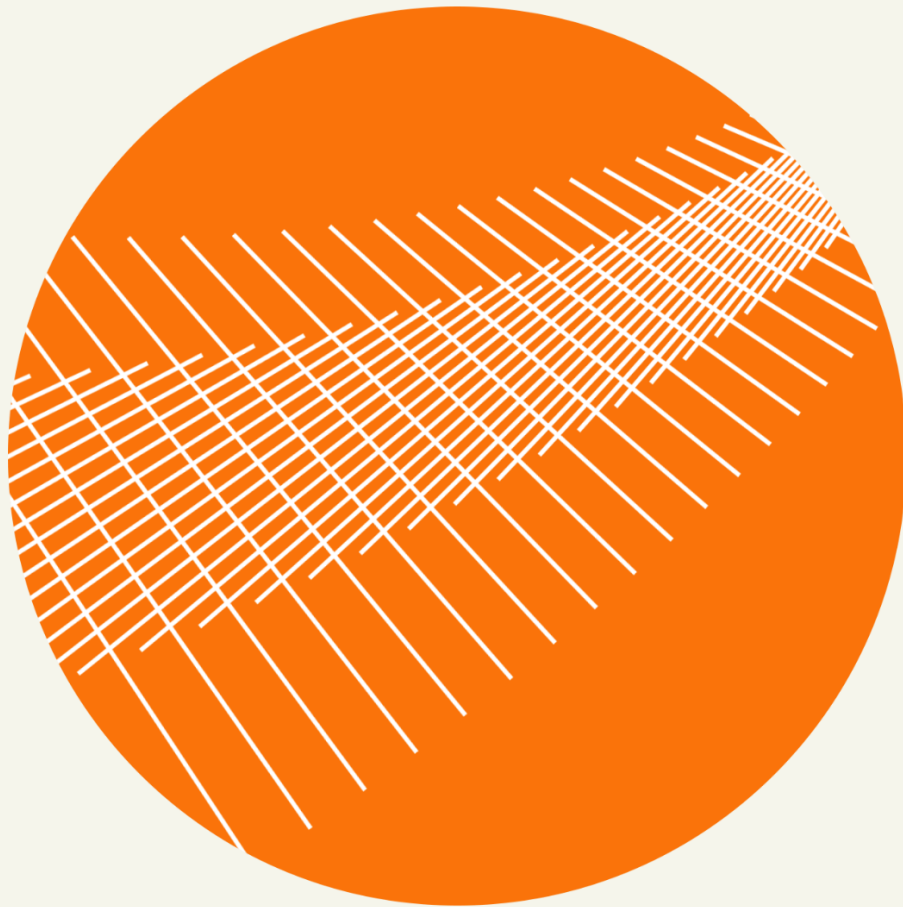


CTDSEC

YOU ARE PROTECTED



TERMS

No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright a CTDSec, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, in any form or by any means such as any information storage, transmission or retrieval system, without prior written permission.

Smart contract security audit

URM FOUNTAIN - PASS

Table of Contents

1.0 Introduction	3
1.1 Project engagement	3
1.2 Disclaimer	3
2.0 Coverage	4
2.1 Target Code and Revision	4
2.2 Attacks made to the contract	5
3.0 Security Issues	7
3.1 High severity issues [1]	7
3.2 Medium severity issues [6]	7
3.3 Low severity issues [0]	10
3.4 Informational Findings [0]	10
4.0 Testing coverage	12
5.0 Annexes	13
6.0 Summary of the audit [PASS 	28

1.0 Introduction

1.1 Project engagement

During September of 2026, URM FOUNTAIN 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 FOUNTAIN provided CTDSec with access to their documentation.

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 FOUNTAIN 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 FOUNTAIN 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:

[urm_fountain.zip \[SHA256\]:](#)

[2173e9465276ce310b711b3752c31ea6d27c98aa640fb1f98f23640302826d59](#)

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.

No	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 [1]

1. Funding Guard Can Consume Late Holder Principal [Fixed

Location:

FountainUrm.sol [line 580 and 808]

Issue:

The vault can account for a bounded amount of interest that the Fortress has not yet transferred. While open, withdrawals are priced against that owed value rather than the URM physically held. An early holder can therefore withdraw funded principal plus part of the unfunded yield allowance, leaving less physical URM for later holders. If Fortress funding later arrives, the remaining claim recovers. If the vault closes or holders exit while funding remains absent, the loss becomes permanent.

Recommendation:

Decide whether principal protection during a funding outage is a protocol guarantee. If it is, open withdrawals must be capped at a pro-rata share of the current URM balance whenever the vault is underfunded. Alternatively, pause ordinary withdrawals while underfunded and allow only a clearly documented pro-rata emergency wind-down.

Fix: The key change is `ratioAssets()` now uses `min(totalAssets(), URM balance)` while the vault is open as well as after closure. Underfunded holders therefore redeem their pro-rata share of funds actually held. `bookValue()` also removes the exiting holders pro-rata share of the total claim, preserving the remaining holders claim if Fortress later funds the shortfall. New deposits are rejected while underfunded, preventing new capital from entering at a distorted share price.

3.2 Medium severity issues [6]

1. CDP Debt Reduction Erases Pending Yield [N/A]

Location:

FountainUrm.sol [line 833 and 914]

Issue:

A repayment in the CDP is applied retroactively to the full interval since the last accrual, erasing previously quoted yield.

Recommendation:

Settle the old debt regime before applying reductions, or apply debt changes only at the next Fortress checkpoint.

2. Transient Withdrawal State Overstates Stored URM [N/A]

Location:

FountainUrm.sol [line 368, 580 and 646]

Issue:

During the Fortress fee callback, the withdrawal has been removed from accounting but remains physically in the vault, so fortressData() temporarily reports it as stored/idle.

Recommendation:

Track pending withdrawals explicitly and exclude them from fortressData() until the user transfer completes.

3. Transient PoolManager Claims Count as Stored URM [N/A]

Location:

FountainUrmHook.sol [line 157 and 201]

Issue:

fortressData() counts PoolManager claims before the router settles the corresponding token payment. The accounting mismatch is reproducible, but downstream Fortress exploitation was not proven.

Recommendation:

Exclude unsettled claims from backing data, or expose them separately as pending liabilities until settlement.

4. Swapper Return Values Can Over-Issue Shares [Fixed

Location:

FountainUrm.sol [line 156]

Issue:

depositUsdc() trusts the swapper's returned URM amount without checking the actual URM balance increase. A short-delivering swapper can cause shares to be minted against missing assets.

Recommendation:

Measure token balance deltas before and after each swap and use the actual received amount.

Fix: depositUsdc() now derives assets from the vault actual URM balance increase, so a swapper that reports 100 URM but transfers only 50 URM mints shares for only the received 50 URM. The vault recorded assets remain equal to its physical URM balance. withdrawUsdc() now pays the user from the actual USDC balance increase. A swapper that reports twice the USDC it delivered cannot drain unrelated USDC already held by the vault.

5. CDP Failure Blocks Exit and Shutdown [N/A]

Location:

FountainUrm.sol [Line 242, 580, 631]

Issue:

A reverting CDP blocks voluntary redemption and prevents the second owner from completing vault closure.

Recommendation:

Store the last valid CDP borrowed value and use it when the live read fails. An emergency mode should freeze additional yield accrual, reject new deposits, and allow two-owner closure plus principal-only redemptions using the cached state.

6. Fortress Failure Blocks Open Withdrawals [Fixed

Location:

FountainURM.sol [line 580 and 646]

Issue:

Every open-state withdrawal must pay a fee through Fortress. A paused or reverting Fortress can therefore freeze normal withdrawals.

Recommendation:

Allow a zero-fee emergency withdrawal path, escrow fees for later collection, or permit withdrawals when Fortress is unavailable.

Fix: The try/catch makes fee forwarding non-blocking: when `payFortress(fee)` reverts, the redemption or deposit can still complete and the fee remains in the Fountain.

Additional recommendations:

Kept fees are not automatically forwarded once Fortress recovers. Decide whether they intentionally remain vault reserves or add a restricted, successful-only `flushKeptFees()` flow.

Emit a `FeesKept(fee)` event so an outage cannot silently accumulate retained protocol fees.

If the plan is to have it proxied/upgradable, append `totalFeesKept` at the end of the storage layout. Inserting it before existing fields shifts downstream slots.

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.*

25

tests passed

0

tests failed

83.91%

FountainUrm lines

80.13%

Hook lines

Contract	Lines	Statements	Branches	Functions
contracts/FountainUrm.sol	83.91% (365/435)	82.97% (458/552)	46.81% (66/141)	85.53% (65/76)
contracts/FountainUrmHook.sol	80.13% (121/151)	80.12% (137/171)	35.42% (17/48)	87.50% (28/32)
TOTAL	82.99% (610/735)	82.21% (707/860)	45.50% (96/211)	84.83% (123/145)

5.0 Annexes

Fountain testing:

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

import {FountainUrm} from "../contracts/FountainUrm.sol";
import {TowerStructs} from "../contracts/TowerStructs.sol";
import {IUrmFortress} from "../contracts/IUrmFortress.sol";

interface Vm {
    function etch(address, bytes calldata) external;
    function prank(address) external;
    function startPrank(address) external;
    function stopPrank() external;
    function warp(uint256) external;
    function expectRevert() external;
    function expectRevert(bytes calldata) external;
}

library Addr {
    address internal constant URM =
0xF0E3349f00972475a5D16721796091AA3B997837;
    address internal constant CDP =
0xAE7F3D2f87970C40f700e6818a6001472108059B;
    address internal constant FORTRESS =
0xd3e9637Cfc255AD3FffA1e88Bd9BA7fa5a76b4Fe;
    address internal constant UDO =
0x268B9Be506E7338a3461256cfe86BAE7205aA9bc;
    address internal constant USDC =
0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913;
    address internal constant SWAPPER =
0x1A81e7C3BF668Dd2D58B6D62D36846664E091cDa;
    address internal constant PM =
0x498581fF718922c3f8e6A244956aF099B2652b2b;
    address internal constant OWNER1 = address(0x1001);
    address internal constant OWNER2 = address(0x1002);
    address internal constant AUTOMATOR = address(0xA770);
    address internal constant ALICE = address(0xA11CE);
```

```

    address internal constant BOB = address(0xB0B);
}

contract MockToken {
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;
    uint256 public totalSupply;
    function approve(address spender, uint256 amount) external returns (bool)
    {
        allowance[msg.sender][spender] = amount;
        return true;
    }
    function transfer(address to, uint256 amount) external returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }
    function transferFrom(address from, address to, uint256 amount) external
returns (bool) {
        if (allowance[from][msg.sender] != type(uint256).max)
allowance[from][msg.sender] -= amount;
        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        return true;
    }
    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
        totalSupply += amount;
    }
    function burn(uint256 amount) external {
        balanceOf[msg.sender] -= amount;
        totalSupply -= amount;
    }
}

contract MockCdp {
    uint256 public borrowed;
    bool public broken;
    function setBorrowed(uint256 value) external { borrowed = value; }
    function setBroken(bool value) external { broken = value; }
    function getBorrowedUrm() external view returns (uint256) {
        require(!broken, "cdp unavailable");
    }
}

```

```

        return borrowed;
    }
}

contract MockOwnership {
    function getOwners() external pure returns (address, address) { return
(Addr.OWNER1, Addr.OWNER2); }
    function getAutomator() external pure returns (address) { return
Addr.AUTOMATOR; }
    function getMultisig() external pure returns (address) { return
address(0xBEEF); }
}

contract MockFortress {
    bool public broken;
    address public observer;
    uint256 public observedStored;
    function setBroken(bool value) external { broken = value; }
    function setObserver(address value) external { observer = value; }
    function payFortress(uint256 amount) external {
        require(!broken, "fortress unavailable");
        if (observer != address(0)) {
            (bool ok, bytes memory data) =
observer.staticcall(abi.encodeWithSignature("fortressData()"));
            if (ok) {
                (, , , observedStored,) = abi.decode(data, (uint256, uint256,
uint256, uint256, uint256));
            }
        }
        MockToken(Addr.URM).transferFrom(msg.sender, address(this), amount);
    }
    function overview() external pure returns (IUrmFortress.FortressOverview
memory r) {
        r.totalUsdcPooled = 1_000_000e6;
    }
}

contract MockSwapper {
    bool public shortOutput;
    function setShortOutput(bool value) external { shortOutput = value; }
    function swapUsdcToUrm(uint256 amount, uint256, uint256) external returns
(uint256) {

```

```

        MockToken(Addr.USDC).transferFrom(msg.sender, address(this), amount);
        uint256 output = amount * 1e12;
        MockToken(Addr.URM).mint(msg.sender, shortOutput ? output / 2 :
output);
        return output;
    }
    function swapUrmToUsdc(uint256 amount, uint256, uint256) external returns
(uint256) {
        MockToken(Addr.URM).transferFrom(msg.sender, address(this), amount);
        MockToken(Addr.USDC).mint(msg.sender, amount / 1e12);
        return amount / 1e12;
    }
}

abstract contract FountainFixture {
    Vm internal constant vm = Vm(address(uint160(uint256(keccak256("hevm
cheat code")))));
    FountainUrm internal vault;
    event log_named_uint(string key, uint256 val);

    function _install() internal {
        vm.warp(1_800_000_000);
        bytes memory token = address(new MockToken()).code;
        vm.etch(Addr.URM, token);
        vm.etch(Addr.USDC, token);
        vm.etch(Addr.CDP, address(new MockCdp()).code);
        vm.etch(Addr.UDO, address(new MockOwnership()).code);
        vm.etch(Addr.FORTRESS, address(new MockFortress()).code);
        vm.etch(Addr.SWAPPER, address(new MockSwapper()).code);
        MockCdp(Addr.CDP).setBorrowed(1_000_000e18);
        vault = new FountainUrm();
        TowerStructs.FountainConfig memory cfg = vault.getState().config;
        cfg.depositOpen = true;
        vm.prank(Addr.AUTOMATOR);
        vault.setConfigs(cfg);
    }

    function _deposit(address who, uint256 assets) internal returns (uint256
shares) {
        MockToken(Addr.URM).mint(who, assets);
        vm.startPrank(who);
        MockToken(Addr.URM).approve(address(vault), type(uint256).max);
    }
}

```

```

    shares = vault.deposit(assets, who);
    vm.stopPrank();
}

function _sync() internal {
    vm.prank(Addr.AUTOMATOR);
    vault.fortressSync();
}

function _close() internal {
    vm.prank(Addr.OWNER1);
    vault.closeTower();
    vm.prank(Addr.OWNER2);
    vault.closeTower();
}

function _abs(uint256 a, uint256 b) internal pure returns (uint256) {
    return a > b ? a - b : b - a; }
}

contract FountainAuditTest is FountainFixture {
    function setUp() public { _install(); }

    function test_CdpDebtDropErasesPreviouslyQuotedAccruedYield() public {
        _deposit(Addr.ALICE, 100_000e18);
        vm.warp(block.timestamp + 4 hours);
        uint256 beforeDrop = vault.totalAssets();
        uint256 principal = vault.getState().state.totalAssets;
        require(beforeDrop > principal, "no accrued interest");
        MockCdp(Addr.CDP).setBorrowed(0);
        require(vault.totalAssets() == principal, "debt drop did not erase
pending yield");
        emit log_named_uint("erased interest wei", beforeDrop - principal);
        _deposit(Addr.BOB, 1e18);
        MockCdp(Addr.CDP).setBorrowed(1_000_000e18);
        require(vault.getState().state.totalInterest == 0, "loss was not
permanent");
    }

    function test_UnfundedGuardDoesNotProtectLateHolderPrincipal() public {
        uint256 aliceShares = _deposit(Addr.ALICE, 5_000e18);
        _deposit(Addr.BOB, 5_000e18);
    }
}

```

```

uint256 principalEach = 4_975e18;
vm.warp(block.timestamp + 31 days);
require(vault.getState().guardBinding, "guard not active");
vm.prank(Addr.ALICE);
uint256 early = vault.redeem(aliceShares, Addr.ALICE, Addr.ALICE);
uint256 remaining = MockToken(Addr.URM).balanceOf(address(vault));
emit log_named_uint("early holder receives wei", early);
emit log_named_uint("late holder balance wei", remaining);
require(early > principalEach, "early holder did not take interest from
principal");
require(remaining < principalEach, "late principal intact");
uint256 bobShares = vault.balanceOf(Addr.BOB);
vm.prank(Addr.BOB);
vm.expectRevert(bytes("bal"));
vault.redeem(bobShares, Addr.BOB, Addr.BOB);
}

function test_ClosedVaultStillDependsOnCdpForVoluntaryExit() public {
    uint256 shares = _deposit(Addr.ALICE, 1_000e18);
    _close();
    MockCdp(Addr.CDP).setBroken(true);
    vm.prank(Addr.ALICE);
    vm.expectRevert(bytes("cdp unavailable"));
    vault.redeem(shares, Addr.ALICE, Addr.ALICE);
    address[] memory holders = new address[](1);
    holders[0] = Addr.ALICE;
    vm.prank(Addr.AUTOMATOR);
    vault.forceReturn(holders);
    require(vault.balanceOf(Addr.ALICE) == 0, "forceReturn fallback
failed");
}

function test_CdpOutageBlocksExitAndSecondOwnerShutdown() public {
    uint256 shares = _deposit(Addr.ALICE, 1_000e18);
    vm.prank(Addr.OWNER1);
    vault.closeTower();
    MockCdp(Addr.CDP).setBroken(true);

    vm.prank(Addr.ALICE);
    vm.expectRevert(bytes("cdp unavailable"));
    vault.redeem(shares, Addr.ALICE, Addr.ALICE);

```

```

    vm.prank(Addr.OWNER2);
    vm.expectRevert(bytes("cdp unavailable"));
    vault.closeTower();
}

function test_WithdrawalFeeCallbackSeesPayoutAsStoredBeforeTransfer()
public {
    _deposit(Addr.ALICE, 1_000e18);
    MockFortress(Addr.FORTRESS).setObserver(address(vault));

    vm.prank(Addr.ALICE);
    vault.withdraw(100e18, Addr.ALICE, Addr.ALICE);

    (, , , uint256 observedStored,) = vault.fortressData();
    uint256 callbackStored = MockFortress(Addr.FORTRESS).observedStored();
    require(callbackStored > 0, "callback did not observe transient stored
balance");
    require(callbackStored > observedStored, "stored balance was not
reduced after payout transfer");
}

function test_SwapperShortReturnOverissuesUsdcDepositShares() public {
    MockToken(Addr.USDC).mint(Addr.ALICE, 100e6);
    MockSwapper(Addr.SWAPPER).setShortOutput(true);

    vm.startPrank(Addr.ALICE);
    MockToken(Addr.USDC).approve(address(vault), 100e6);
    uint256 shares = vault.depositUsdc(100e6, 0, Addr.ALICE);
    vm.stopPrank();

    require(shares > 0, "short-return deposit did not mint shares");
    require(vault.getState().state.totalAssets >
MockToken(Addr.URM).balanceOf(address(vault)), "short return was
detected");
}

function test_ZeroCdpDebtDoesNotBlockPrincipalExit() public {
    uint256 shares = _deposit(Addr.ALICE, 1_000e18);
    MockCdp(Addr.CDP).setBorrowed(0);
    vm.prank(Addr.ALICE);
    require(vault.redeem(shares, Addr.ALICE, Addr.ALICE) > 0, "exit
blocked");
}

```

```

}

function test_CannotWithdrawAnotherHoldersSharesWithoutAllowance() public
{
    uint256 shares = _deposit(Addr.ALICE, 1_000e18);
    vm.prank(Addr.BOB);
    vm.expectRevert(bytes("auth"));
    vault.redeem(shares, Addr.BOB, Addr.ALICE);
}

function test_CloseRequiresTwoDifferentOwners() public {
    vm.startPrank(Addr.OWNER1);
    vault.closeTower();
    vault.closeTower();
    vm.stopPrank();
    require(!vault.getState().state.closed, "single owner closed vault");
    vm.prank(Addr.OWNER2);
    vault.closeTower();
    require(vault.getState().state.closed, "second owner did not close");
}

function test_ExpiredCloseProposalCannotExecute() public {
    vm.prank(Addr.OWNER1);
    vault.closeTower();
    vm.warp(block.timestamp + 1 days + 1);
    vm.prank(Addr.OWNER2);
    vault.closeTower();
    require(!vault.getState().state.closed, "expired proposal executed");
}

function test_NewDepositCannotImmediatelyRaiseStoredCheckpoint() public {
    uint256 startedAt = block.timestamp;
    _sync();
    _deposit(Addr.ALICE, 100_000e18);
    (, , , uint256 stored,) = vault.fortressData();
    require(stored == 0, "fresh principal immediately excluded");
    vm.warp(startedAt + 6 hours);
    _sync();
    require(vault.getState().state.storedCheckpoint == 0, "not held for
full checkpoint interval");
    vm.warp(startedAt + 12 hours);
    _sync();
}

```

```

    require(vault.getState().state.storedCheckpoint >= 99_500e18, "mature
deposit never counted");
}

function test_DonationCannotInflateSharePrice() public {
    _deposit(Addr.ALICE, 1e18);
    uint256 quote = vault.previewDeposit(100e18);
    MockToken(Addr.URM).mint(address(vault), 1_000_000e18);
    require(vault.previewDeposit(100e18) == quote, "donation moved share
price");
}

function test_UsdcCallerSlippageFloorRevertsAtomically() public {
    MockToken(Addr.USDC).mint(Addr.ALICE, 100e6);
    vm.startPrank(Addr.ALICE);
    MockToken(Addr.USDC).approve(address(vault), 100e6);
    vm.expectRevert(bytes("minOut"));
    vault.depositUsdc(100e6, 100e18, Addr.ALICE);
    vm.stopPrank();
    require(MockToken(Addr.USDC).balanceOf(Addr.ALICE) == 100e6, "input was
not restored");
}

function testFuzz_PreviewDepositAndMintMatchExecution(uint96 input)
public {
    uint256 amount = 1e18 + uint256(input) % 100_000e18;
    uint256 quoted = vault.previewDeposit(amount);
    require(_deposit(Addr.ALICE, amount) == quoted, "deposit preview
mismatch");
    uint256 shares = amount;
    uint256 cost = vault.previewMint(shares);
    MockToken(Addr.URM).mint(Addr.BOB, cost);
    vm.startPrank(Addr.BOB);
    MockToken(Addr.URM).approve(address(vault), cost);
    require(vault.mint(shares, Addr.BOB) == cost, "mint preview mismatch");
    vm.stopPrank();
}

function testFuzz_MaxRedeemIsExecutable(uint96 input, uint32 elapsed)
public {
    uint256 amount = 1e18 + uint256(input) % 1_000_000e18;
    _deposit(Addr.ALICE, amount);

```

```

    vm.warp(block.timestamp + uint256(elapsed) % 60 days);
    uint256 maximum = vault.maxRedeem(Addr.ALICE);
    if (maximum == 0) return;
    vm.prank(Addr.ALICE);
    uint256 assets = vault.redeem(maximum, Addr.ALICE, Addr.ALICE);
    require(assets > 0, "maximum redeemed zero");
}

function testFuzz_DepositRedeemRoundTripCannotProfit(uint96 input) public
{
    uint256 amount = 1e18 + uint256(input) % 10_000e18;
    uint256 shares = _deposit(Addr.ALICE, amount);
    vm.prank(Addr.ALICE);
    uint256 received = vault.redeem(shares, Addr.ALICE, Addr.ALICE);
    require(received < amount, "profitable same-block round trip");
}
}

```

Hook Testing:

```

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

import {FountainFixture, Addr, MockToken} from "../FountainAudit.t.sol";
import {FountainUrmHook} from "../contracts/FountainUrmHook.sol";
import {PoolManager} from "@uniswap/v4-core/src/PoolManager.sol";
import {IPoolManager} from
"@uniswap/v4-core/src/interfaces/IPoolManager.sol";
import {IUnlockCallback} from
"@uniswap/v4-core/src/interfaces/callback/IUnlockCallback.sol";
import {IHooks} from "@uniswap/v4-core/src/interfaces/IHooks.sol";
import {PoolKey} from "@uniswap/v4-core/src/types/PoolKey.sol";
import {SwapParams} from "@uniswap/v4-core/src/types/PoolOperation.sol";
import {Currency} from "@uniswap/v4-core/src/types/Currency.sol";
import {BalanceDelta, BalanceDeltaLibrary} from
"@uniswap/v4-core/src/types/BalanceDelta.sol";
import {TickMath} from "@uniswap/v4-core/src/libraries/TickMath.sol";

contract AuditRouter is IUnlockCallback {
    using BalanceDeltaLibrary for BalanceDelta;
    IPoolManager internal constant manager = IPoolManager(Addr.PM);
}

```

```

uint256 public observedStored;

function run(PoolKey memory key, bool zeroForOne, int256 amount, uint256
count, bool payInput)
    external returns (bytes memory)
{
    return manager.unlock(abi.encode(key, zeroForOne, amount, count,
payInput, msg.sender, address(0)));
}

function runObserved(PoolKey memory key, bool zeroForOne, int256 amount,
uint256 count, bool payInput, address observer)
    external returns (bytes memory)
{
    return manager.unlock(abi.encode(key, zeroForOne, amount, count,
payInput, msg.sender, observer));
}

function unlockCallback(bytes calldata data) external returns (bytes
memory) {
    require(msg.sender == Addr.PM, "auth");
    (PoolKey memory key, bool zeroForOne, int256 amount, uint256 count,
bool payInput, address recipient, address observer) =
        abi.decode(data, (PoolKey, bool, int256, uint256, bool, address,
address));
    int256 delta0;
    int256 delta1;
    for (uint256 i; i < count; ++i) {
        BalanceDelta delta = manager.swap(key, SwapParams({
            zeroForOne: zeroForOne,
            amountSpecified: amount,
            sqrtPriceLimitX96: zeroForOne ? TickMath.MIN_SQRT_PRICE + 1 :
TickMath.MAX_SQRT_PRICE - 1
        })), "");
        delta0 += delta.amount0();
        delta1 += delta.amount1();
    }
    if (observer != address(0)) {
        (bool ok, bytes memory result) =
observer.staticcall(abi.encodeWithSignature("fortressData()"));
        require(ok, "observer");
        (, , , observedStored,) = abi.decode(result, (uint256, uint256,

```

```

uint256, uint256, uint256));
    }
    _resolve(key.currency0, delta0, payInput, recipient);
    _resolve(key.currency1, delta1, payInput, recipient);
    return abi.encode(delta0, delta1);
}

function _resolve(Currency currency, int256 delta, bool payInput, address
recipient) internal {
    if (delta > 0) manager.take(currency, recipient, uint256(delta));
    else if (delta < 0 && payInput) {
        manager.sync(currency);
        MockToken(Currency.unwrap(currency)).transfer(Addr.PM,
uint256(-delta));
        manager.settle();
    }
}
}

contract HookAuditTest is FountainFixture {
    FountainUrmHook internal hook;
    AuditRouter internal router;
    PoolKey internal key;

    // Execute real initcode at the fixed address so both constructor storage
and immutables are accurate.
    function _deployAt(address target, bytes memory initcode) internal {
        vm.etch(target, initcode);
        (bool ok, bytes memory runtime) = target.call("");
        require(ok && runtime.length > 0, "deployment failed");
        vm.etch(target, runtime);
    }

    function setUp() public {
        _install();
        _deployAt(Addr.PM, abi.encodePacked(type(PoolManager).creationCode,
abi.encode(address(this))));
        address hookAddr = address(0x2888);
        _deployAt(hookAddr,
abi.encodePacked(type(FountainUrmHook).creationCode,
abi.encode(address(vault), 100_000e18)));
        hook = FountainUrmHook(hookAddr);
    }
}

```

```

MockToken(Addr.URM).mint(hookAddr, 100_000e18);
router = new AuditRouter();
bool wrapZeroForOne = hook.wrapZeroForOne();
key = PoolKey({
    currency0: Currency.wrap(wrapZeroForOne ? Addr.URM : address(vault)),
    currency1: Currency.wrap(wrapZeroForOne ? address(vault) : Addr.URM),
    fee: 0,
    tickSpacing: 1,
    hooks: IHooks(hookAddr)
});
IPoolManager(Addr.PM).initialize(key, uint160(1 << 96));
}

function _collect() internal {
    vm.prank(Addr.AUTOMATOR);
    hook.collect();
}

function test_WrapSettlesRouterPaymentAndRestoresFloat() public {
    MockToken(Addr.URM).mint(address(router), 1_000e18);
    router.run(key, hook.wrapZeroForOne(), -int256(1_000e18), 1, true);
    require(vault.balanceOf(address(this)) == 995e18, "unexpected wrap
output");
    require(hook.getState().urmClaims == 1_000e18, "claim missing");
    require(MockToken(Addr.URM).balanceOf(address(hook)) == 99_000e18,
"float not advanced");
    _collect();
    require(hook.getState().urmClaims == 0, "claim not swept");
    require(MockToken(Addr.URM).balanceOf(address(hook)) == 100_000e18,
"float not replenished");
}

function
test_TransientPoolManagerClaimAppearsStoredBeforeRouterSettlement() public
{
    MockToken(Addr.URM).mint(address(router), 1_000e18);
    router.runObserved(key, hook.wrapZeroForOne(), -int256(1_000e18), 1,
true, address(hook));

    require(router.observedStored() == 100_000e18, "claim was not reported
as stored");
    require(MockToken(Addr.URM).balanceOf(address(hook)) == 99_000e18,

```

```

"physical float did not decrease");
    require(hook.getState().urmClaims == 1_000e18, "claim was not
recorded");
}

function test_UnwrapClaimsRedeemAtNextCollect() public {
    _deposit(address(router), 1_000e18);
    uint256 shares = vault.balanceOf(address(router));
    uint256 quote = vault.previewRedeem(shares);
    router.run(key, !hook.wrapZeroForOne(), -int256(shares), 1, true);
    require(MockToken(Addr.URM).balanceOf(address(this)) == quote, "wrong
unwrap amount");
    require(hook.getState().furmClaims == shares, "missing fURM claim");
    _collect();
    require(hook.getState().furmClaims == 0 &&
vault.balanceOf(address(hook)) == 0, "claim not redeemed");
    require(MockToken(Addr.URM).balanceOf(address(hook)) == 100_000e18,
"float lost value");
    require(vault.getState().state.windowWithdrawn == quote, "redemption
not charged to window");
}

function test_MultiSwapSameUnlockBypassesGlobalWithdrawalWindow() public
{
    _deposit(address(router), 100_000e18);
    uint256 cap = vault.windowRemaining();
    require(cap == 10_000e18, "unexpected default cap");
    router.run(key, !hook.wrapZeroForOne(), -int256(9_000e18), 10, true);
    uint256 received = MockToken(Addr.URM).balanceOf(address(this));
    emit log_named_uint("vault daily cap wei", cap);
    emit log_named_uint("same-unlock payout wei", received);
    require(received == 89_550e18 && received > 8 * cap, "window bypass not
reproduced");
    require(vault.windowRemaining() == cap, "swaps consumed vault
allowance");
    require(hook.getState().furmClaims == 90_000e18, "wrong unsettled
claim");
    _collect();
    require(vault.windowRemaining() == 0, "collect did not use window");
    require(vault.balanceOf(address(hook)) > 79_000e18, "window did not
strand hook inventory");
}

```

```

function test_UnsettledInputRevertsWholeSwap() public {
    uint256 floatBefore = MockToken(Addr.URM).balanceOf(address(hook));
    bool wrapDirection = hook.wrapZeroForOne();
    vm.expectRevert();
    router.run(key, wrapDirection, -int256(1_000e18), 1, false);
    require(MockToken(Addr.URM).balanceOf(address(hook)) == floatBefore,
"float drained without payment");
    require(vault.totalSupply() == 0, "failed swap left minted shares");
}

function test_ExactOutputIsRejected() public {
    bool wrapDirection = hook.wrapZeroForOne();
    vm.expectRevert();
    router.run(key, wrapDirection, int256(1e18), 1, true);
}

function test_PoolMustHaveZeroFee() public {
    PoolKey memory invalidKey = key;
    invalidKey.fee = 500;
    vm.expectRevert();
    IPoolManager(Addr.PM).initialize(invalidKey, uint160(1 << 96));
}

function test_NonManagerCannotInvokeUnlockCallback() public {
    vm.expectRevert();
    hook.unlockCallback("");
}

function test_CloseBurnsFloatAndRejectsNewSwaps() public {
    vm.prank(Addr.OWNER1);
    hook.closeTower();
    require(MockToken(Addr.URM).balanceOf(address(hook)) == 0, "close did
not burn float");
    require(hook.getState().state.closed, "hook not closed");
    MockToken(Addr.URM).mint(address(router), 1_000e18);
    bool wrapDirection = hook.wrapZeroForOne();
    vm.expectRevert();
    router.run(key, wrapDirection, -int256(1_000e18), 1, true);
}
}
}
}

```

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	1					1
Medium	6		4			2
Low						
Informational						