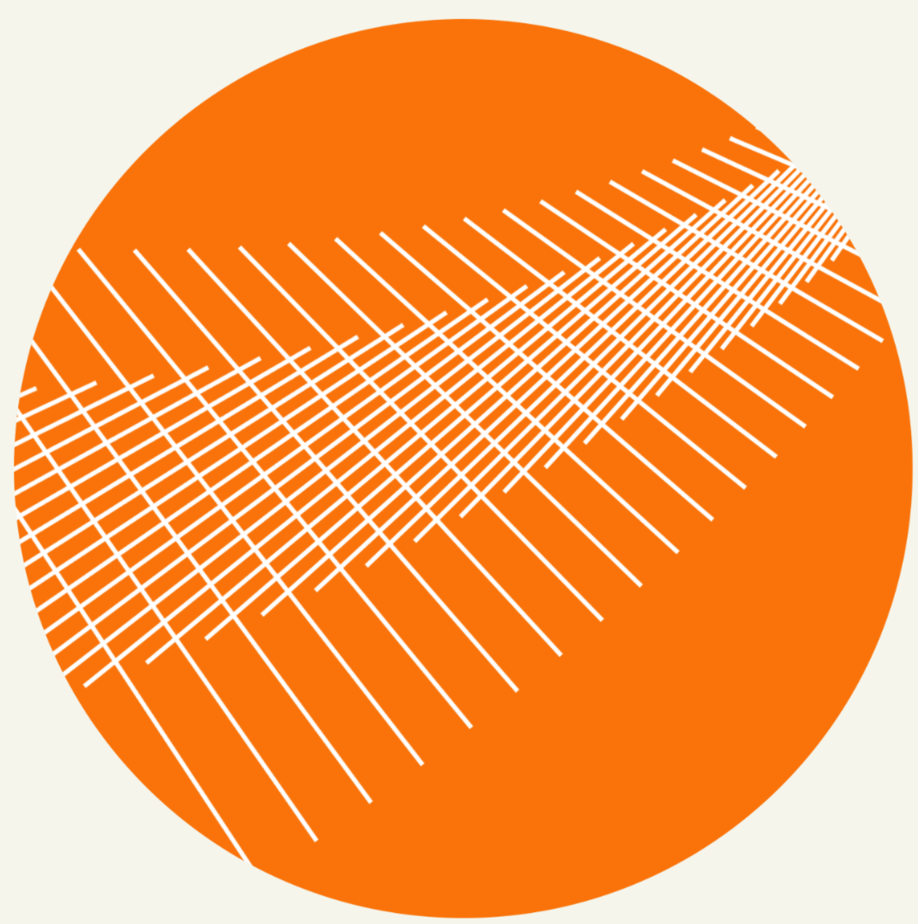
CTDSEC

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 FORTRESS - 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 [3] 7

3.2 Medium severity issues [2] 9

3.3 Low severity issues [0] 10

3.4 Informational Findings [0] 11

4.0 Testing coverage **12**

5.0 Annexes **14**

6.0 Summary of the audit [PASS ✅] **50**

Copyright © ***CTDSec*** - All rights reserved

1.0 Introduction

1.1 Project engagement

During May of 2026, URM FORTRESS 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 FORTRESS 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 FORTRESS team put in place a bug bounty program to encourage further and active analysis of the smart contract.

Copyright © ***CTDSec*** - All rights reserved

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the URM FORTRESS 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_fortress_v3.zip [SHA256]:

c848bcba8a0ef464455923a45a4f0b3b52cdcf6010d1eab93b6eb23fb0b9bb55

Fixed version: contracts_post_audit.zip [sha256]

9fcf92b7764998ae923f8b921362a7d242684d0ff9421d586de4392fb1d53b7f

Copyright © ***CTDSec*** - All rights reserved

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.

Copyright © ***CTDSec*** - All rights reserved

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

Copyright © ***CTDSec*** - All rights reserved

3.0 Security Issues

3.1 High severity issues [3]

1. Above-Peg Defense Can Self-DoS Due to Prior URM Consumption [Partially fixed ✅]

Location:

contracts/UrmFortress.sol:378

contracts/UrmFortress.sol:382-386

contracts/UrmFortress.sol:391

contracts/UrmFortress.sol:575

contracts/UrmFortress.sol:593

Issue: pegDefenseOverview() calculates the amount of URM available for above-peg defense before executing additional actions such as syncing or liquidity increases. However, these intermediate operations may consume part of the URM balance before the actual defense sale occurs. As a result, the subsequent abovePegDefense() call attempts to sell more URM than remains available, causing the transaction to revert and preventing peg stabilization.

Recommendation: Prevent balance inconsistency by:

Reserving pdo.urmToSell before executing any URM-consuming actions

Executing the above-peg sale before liquidity or sync operations

Recomputing balances and defense parameters immediately before performing the sale

Fix: now rechecks getUrmBalance() >= pdo.urmToSell before selling, so the old revert is avoided. However, if sync/liquidity consumes the reserved URM, above-peg defense is skipped after cooldown is consumed.

2. Exact-Output Multi-Hop Swaps Consume Entire Maximum Input Before Refund [Fixed]

Location:

contracts/UrmSwapper.sol:179 (swapRageToExactUrm)

Copyright © ***CTDSec*** - All rights reserved

contracts/UrmSwapper.sol:183

contracts/UrmSwapper.sol:203 (swapHestiaToExactUrm)

contracts/UrmSwapper.sol:206

contracts/UrmSwapper.sol:223 (swapCircleToExactUrm)

contracts/UrmSwapper.sol:226-227

Issue: The exact-output swap functions incorrectly consume the entire user-provided maximum input amount before determining the actual amount required for the exact URM output. Excess funds are only refunded after additional swaps back into the original asset. This design unnecessarily exposes users to additional swap fees, slippage, price impact, and sandwich attack risks on the refunded surplus amount.

Recommendation: Implement a true exact-output multi-hop routing mechanism using amountInMaximum, ensuring that:

- Only the required input amount is consumed
- Unused input is refunded directly without additional swaps
- The entire maximum input is never converted before determining the required amount

Fix: Exact-output functions were removed from UrmSwapper.sol and IUrmSwapper. Searches for Exact, swapToExactOutput, and exact-output function names returned no matches.

3. Slippage Values Below 50 Disable Price-Limit Protection [Partially fixed ✅]

Location:

contracts/UrmSwapper.sol:445 (executeSwap)

contracts/UrmSwapper.sol:452

contracts/UrmSwapper.sol:454

contracts/UrmSwapper.sol:183

contracts/UrmSwapper.sol:206

contracts/UrmSwapper.sol:226-227

contracts/UrmSwapper.sol:232-233

Issue: The executeSwap() function initializes the swap price limit using the extreme Uniswap V3 min/max values and only applies slippage protection when slippage >= 50. Consequently, values between 0 and 49

Copyright © ***CTDSec*** - All rights reserved

effectively disable price-limit protection entirely, allowing swaps to execute at bad rates despite users expecting tighter constraints.

Recommendation: Ensure slippage protection is enforced for all slippage values, including zero, by:

- Applying price-limit calculations regardless of slippage value
- Using amount-based protections such as amountOutMinimum or amountInMaximum
- Rejecting unsupported zero-slippage configurations unless implemented as a strict no-slippage bound

Fix: UrmSwapper.sol changed the guard to if (slippage > 0), so 1-49 bps no longer use full min/max range. But slippage = 0 is still full range, and the integer swapSqrt() math remains coarse: 49/50/100 bps all allow about 1.99% movement in one direction.

3.2 Medium severity issues [2]

1. Disabled Sync Configuration Still Executes Pre-Sync External Calls [Fixed ✅]

Location:

contracts/UrmFortress.sol:549

contracts/UrmFortress.sol:551-559

contracts/UrmFortress.sol:384

Issue: Although CONFIG.syncContract disables the main sync() execution flow, the contract still performs fortressPreSync() external calls on registered sync-enabled contracts during pegDefenseOverview(). A reverting or unavailable external contract can therefore block pegDefense() even when synchronization is intended to be disabled.

Recommendation: Skip all pre-sync external calls when CONFIG.syncContract == false. Additionally, isolate external sync calls using try/catch to prevent a single failing contract from blocking unrelated peg-defense functionality

Fix: UrmFortress now wraps the pre-sync loop in if (CONFIG.syncTowers), so disabled sync no longer calls tower fortressPreSync().

2. Public No-Op pegDefense() Can Consume The Cooldown [Fixed ✅]

Location:

UrmFortress.sol

pegDefense() only requires pdo.pegDefenseAvailable, then immediately sets STATE.nextPegDefenseTime at contracts/UrmFortress.sol:379 and contracts/UrmFortress.sol:380.

It does not require pdo.required.

pegDefenseOverview() computes pdo.required at contracts/UrmFortress.sol:587, but the value is not enforced.

Issue: When pegDefensePublic is enabled, any address can call pegDefense() as soon as the cooldown has elapsed. If no action is currently required, the function still advances STATE.nextPegDefenseTime, consuming the two-minute global cooldown. An attacker can call during a no-op state and delay a real defense that becomes necessary immediately afterward.

Recommendation: Add require(pdo.required, "noop") before setting STATE.nextPegDefenseTime.

Refine pdo.required so CONFIG.syncContract alone does not make a no-op sync appear required; use pdo.contractSyncRequested or actual requested URM instead.

Only update STATE.nextPegDefenseTime after a state-changing action executes, or maintain separate cooldowns for public no-op/sync/defense paths.

Fix: UrmFortress.sol line 354 now requires pdo.required before setting STATE.nextPegDefenseTime, and line 569 computes required from real requested actions, not just the master sync flag.

3.3 Low severity issues [0]

No low severity issues were found.

3.4 Informational Findings [0]

No informational severity issues were found

Copyright © ***CTDSec*** - All rights reserved

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.*

```
Hardhat confirmed vulnerability tests
npx hardhat test



  Hardhat confirmed UrmFortress/UrmSwapper vulnerabilities
    √ HIGH: non-custodied URM/USDC NFT can receive Fortress liquidity and be drained by the NFT owner (244ms)
    √ HIGH: above-peg defense reverts when prior liquidity increase consumes reserved URM (219ms)
    √ MEDIUM: syncContract=false still calls fortressPreSync and can block pegDefense (207ms)
    √ HIGH: exact-output RAGE/HESTIA/CIRCLE swaps sell the entire max input before refunding surplus (263ms)
    √ HIGH: slippage=0 still accepts catastrophic first-leg execution in exact-output routes (215ms)


  5 passing (2s)
```

Copyright © ***CTDSec*** - All rights reserved

```
Command: forge coverage test\UrmSwapperExactOutputCoverage.t.sol
Options: --offline --ir-minimum --skip Fortress-heavy files --exclude-tests

Solc 0.8.26 finished in 4.44s
Compiler run successful!
Analysing contracts...
Running tests...

Ran 4 tests for test/UrmSwapperExactOutputCoverage.t.sol:UrmSwapperExactOutputCoverageTest
[PASS] testCoverage_circleExactOutputSellsMaxInput() (gas: 18325493)
[PASS] testCoverage_hestiaExactOutputSellsMaxInput() (gas: 18231604)
[PASS] testCoverage_rageExactOutputSellsMaxInput() (gas: 18236929)
[PASS] testCoverage_zeroSlippageStillAllowsBadFirstLeg() (gas: 18139251)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 5.08ms (15.33ms CPU time)

Ran 1 test suite in 6.23ms (5.08ms CPU time): 4 tests passed, 0 failed, 0 skipped (4 total tests)

?--------------------------------+-----------------+------------------+-----------------+------------
| File                           | % Lines         | % Statements     | % Branches      | % Funcs
+===================================================================================================
| contracts/ERC20.sol            | 0.00% (0/41)    | 0.00% (0/35)     | 0.00% (0/5)     | 0.00% (0/9)
|--------------------------------+-----------------+------------------+-----------------+------------
| contracts/FullMath.sol         | 0.00% (0/34)    | 0.00% (0/35)     | 0.00% (0/8)     | 0.00% (0/2)
|--------------------------------+-----------------+------------------+-----------------+------------
| contracts/LiquidityAmounts.sol | 0.00% (0/33)    | 0.00% (0/39)     | 0.00% (0/16)    | 0.00% (0/7)
|--------------------------------+-----------------+------------------+-----------------+------------
| contracts/OracleLibrary.sol    | 0.00% (0/53)    | 0.00% (0/68)     | 0.00% (0/16)    | 0.00% (0/6)
|--------------------------------+-----------------+------------------+-----------------+------------
| contracts/ReentrancyGuard.sol  | 0.00% (0/4)     | 0.00% (0/3)      | 0.00% (0/2)     | 0.00% (0/1)
|--------------------------------+-----------------+------------------+-----------------+------------
| contracts/TickMath.sol         | 0.00% (0/115)   | 0.00% (0/160)    | 0.00% (0/24)    | 0.00% (0/2)
|--------------------------------+-----------------+------------------+-----------------+------------
| contracts/UrmSwapper.sol       | 35.38% (92/260) | 36.68% (117/319) | 11.25% (18/160) | 25.00% (8/32
|--------------------------------+-----------------+------------------+-----------------+------------
| Total                          | 17.04% (92/540) | 17.75% (117/659) | 7.79% (18/231)  | 13.56% (8/59
?--------------------------------+-----------------+------------------+-----------------+------------
```

Copyright © ***CTDSec*** - All rights reserved

5.0 Annexes

Hardat:

```
const { expect } = require("chai");

const { ethers, network } = require("hardhat");


const USDC = "0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913";

const WETH = "0x4200000000000000000000000000000000000006";

const RAGE = "0xc0df50143EA93AeC63e38A6ED4E92B378079eA15";

const HESTIA = "0xBC7755a153E852CF76cCCDdb4C2e7c368f6259D8";

const CIRCLE = "0x5baBfc2F240bc5De90Eb7e19D789412dB1dEc402";

const HESTIA_USDC_POOL = "0x1B39fC4C93EfbE733B8D2770bcfaa46885d5343a";

const CIRCLE_WETH_POOL = "0xDA679706FF21114AC9faC5198BfF24543F357a16";

const RAGE_USDC_POOL = "0xd474B32a5a2BF93453996287D361a00f661E04FF";

const USDC_WETH_POOL = "0xb2cc224c1c9feE385f8ad6a55b4d94E92359DC59";

const RAGE_DEPOT = "0x6101c37bB1b3A2A9D8dF14B5a6A97047E7B1628e";

const RAGE_ORACLE = "0x9B8a45C4A0fBD44158480D9b4B41e0bdCA42874C";

const RCE = "0x4C15F778Ab59F25D5dFD2dD508236a25eD2813fe";

const POSITION_MANAGER = "0x03a520b32C04BF3bEEf7BEb72E919cf822Ed34f1";

const FACTORY = "0x33128a8fC17869897dcE68Ed026d694621f6FDfD";

const RBP = "0xff70Cd1E1931372F869c936582a7F42e49B6DA4c";

const NFT_ID_URM = 1n;

const NFT_ID_RAGE = 4095881n;
```

Copyright © ***CTDSec*** - All rights reserved

```
const E18 = 10n ** 18n;

const BPS = 10_000n;


async function expectRevert(promise, reason) {

  try {

    await promise;

  } catch (err) {

    expect(err.message).to.include(reason);

    return;

  }

  throw new Error(`Expected revert containing ${reason}`);

}


async function expectAnyRevert(promise) {

  try {

    await promise;

  } catch (_err) {

    return;

  }

  throw new Error("Expected transaction to revert");

}


async function setCodeFrom(contractName, target) {

  const factory = await ethers.getContractFactory(contractName);
```

Copyright © ***CTDSec*** - All rights reserved

```
  const implementation = await factory.deploy();

  await implementation.waitForDeployment();

  const code = await ethers.provider.getCode(await implementation.getAddress());

  await network.provider.send("hardhat_setCode", [target, code]);

  return ethers.getContractAt(contractName, target);

}


async function installToken(target, symbol, decimals = 18) {

  const token = await setCodeFrom("HHMockERC20", target);

  await token.initialize(symbol, symbol, decimals);

  return token;

}


async function installPool(target, token0, token1) {

  const pool = await setCodeFrom("HHMockPool", target);

  await pool.initialize(token0, token1);

  return pool;

}


async function installCommonAddressMocks(signers) {

  await installToken(USDC, "USDC", 6);

  await installToken(RAGE, "RAGE");

  await installToken(HESTIA, "HESTIA");

  await installToken(CIRCLE, "CIRCLE");
```

```
  await installPool(RAGE_USDC_POOL, RAGE, USDC);


  await setCodeFrom("HHMockDepot", RAGE_DEPOT);
  await setCodeFrom("HHMockRageOracle", RAGE_ORACLE);
  const rce = await setCodeFrom("HHMockRce", RCE);
  await setCodeFrom("HHMockFactory", FACTORY);
  await setCodeFrom("HHMockPositionManager", POSITION_MANAGER);
  await setCodeFrom("HHMockRbp", RBP);


  await rce.initialize(
    signers.owner1.address,
    signers.owner2.address,
    signers.automator.address,
    signers.multisig.address,
    RBP
  );
}


async function deployFortressEnv(urmPrice, signers) {
  await installCommonAddressMocks(signers);


  const Urm = await ethers.getContractFactory("HHMockUrm");
  const urm = await Urm.deploy();
  await urm.waitForDeployment();
```

Copyright © ***CTDSec*** - All rights reserved

```
  await urm.initializeUrm();


  const Oracle = await ethers.getContractFactory("HHMockOracle");

  const oracle = await Oracle.deploy();

  await oracle.waitForDeployment();

  await oracle.setUrmPrice(urmPrice);


  const Swapper = await ethers.getContractFactory("HHMockSwapper");

  const swapper = await Swapper.deploy(await urm.getAddress(), USDC);

  await swapper.waitForDeployment();


  const urmPoolAddress = "0x0000000000000000000000000000000000100101";

  const urmPool = await installPool(urmPoolAddress, USDC, await
urm.getAddress());

  const factory = await ethers.getContractAt("HHMockFactory", FACTORY);

  await factory.setPool(urmPoolAddress);


  const positionManager = await ethers.getContractAt("HHMockPositionManager",
POSITION_MANAGER);

  await positionManager.configurePosition(NFT_ID_URM, signers.nftOwner.address,
USDC, await urm.getAddress(), 100, -60, 60, 0);

  await positionManager.configurePosition(NFT_ID_RAGE, signers.owner1.address,
RAGE, USDC, 100, -60, 60, 0);


  const Fortress = await ethers.getContractFactory("UrmFortress");
```

Copyright © ***CTDSec*** - All rights reserved

```
  const fortress = await Fortress.deploy(await urm.getAddress(), await
swapper.getAddress(), await oracle.getAddress(), urmPoolAddress, NFT_ID_URM);

  await fortress.waitForDeployment();


  return { fortress, urm, oracle, swapper, positionManager, urmPool };

}


async function addFortressContract(fortress, contractAddress, signers) {

  await fortress.connect(signers.owner1).addContract(contractAddress, 1);

  await fortress.connect(signers.owner2).addContract(contractAddress, 1);

}


function emptyContractConfigs() {

  return Array.from({ length: 9 }, () => ({

    recipientPercent: 0,

    syncEnabled: false,

    backingEnabled: false

  }));

}


function fortressConfig(syncContract) {

  return {

    defenseSize: 50_000_000,

    assetsValuePercent: 75,
```

Copyright © ***CTDSec*** - All rights reserved

```
    ragePoolPercent: 50,

    twap: 180,

    slippage: 500,

    abovePegThreshold: 1_005_000,

    belowPegThreshold: 999_000,

    belowPegThreshold2: 995_000,

    abovePegDelay: 120,

    belowPegDelay: 120,

    abovePegBuy: 50,

    pegDefenseTarget: 0,

    requestSupplyPercent: 10,

    requestSupplyMin: 50_000n * E18,

    burnSupplyMin: 100_000n * E18,

    pegDefensePublic: false,

    syncContract,

    collectFeesDelay: 7 * 24 * 60 * 60,

    liquidityIncrease: true

  };

}


async function deploySwapperEnv(signers) {

  await installToken(USDC, "USDC");

  await installToken(RAGE, "RAGE");

  await installToken(HESTIA, "HESTIA");
```

```
  await installToken(CIRCLE, "CIRCLE");

  await installToken(WETH, "WETH");


  const Urm = await ethers.getContractFactory("HHMockUrm");

  const urm = await Urm.deploy();

  await urm.waitForDeployment();

  await urm.initializeUrm();


  const urmPoolAddress = "0x0000000000000000000000000000000000200202";

  await installPool(urmPoolAddress, USDC, await urm.getAddress());

  await installPool(RAGE_USDC_POOL, RAGE, USDC);

  await installPool(HESTIA_USDC_POOL, HESTIA, USDC);

  await installPool(CIRCLE_WETH_POOL, CIRCLE, WETH);

  await installPool(USDC_WETH_POOL, WETH, USDC);


  await (await ethers.getContractAt("HHMockERC20", USDC)).mint(RAGE_USDC_POOL,
10_000n * E18);

  await (await ethers.getContractAt("HHMockERC20", RAGE)).mint(RAGE_USDC_POOL,
10_000n * E18);

  await (await ethers.getContractAt("HHMockERC20", USDC)).mint(HESTIA_USDC_POOL,
10_000n * E18);

  await (await ethers.getContractAt("HHMockERC20",
HESTIA)).mint(HESTIA_USDC_POOL, 10_000n * E18);

  await (await ethers.getContractAt("HHMockERC20", WETH)).mint(CIRCLE_WETH_POOL,
10_000n * E18);
```

Copyright © ***CTDSec*** - All rights reserved

```
  await (await ethers.getContractAt("HHMockERC20",
CIRCLE)).mint(CIRCLE_WETH_POOL, 10_000n * E18);

  await (await ethers.getContractAt("HHMockERC20", USDC)).mint(USDC_WETH_POOL,
10_000n * E18);

  await (await ethers.getContractAt("HHMockERC20", WETH)).mint(USDC_WETH_POOL,
10_000n * E18);

  await urm.mint(urmPoolAddress, 10_000n * E18);


  const rce = await setCodeFrom("HHMockRce", RCE);

  await setCodeFrom("HHMockRbp", RBP);

  await rce.initialize(signers.owner1.address, signers.owner2.address,
signers.automator.address, signers.multisig.address, RBP);


  const Oracle = await ethers.getContractFactory("HHMockOracle");

  const oracle = await Oracle.deploy();

  await oracle.waitForDeployment();

  const Swapper = await ethers.getContractFactory("UrmSwapper");

  const swapper = await Swapper.deploy(await urm.getAddress(), urmPoolAddress,
await oracle.getAddress());

  await swapper.waitForDeployment();


  return { swapper, urm };

}


describe("Hardhat confirmed UrmFortress/UrmSwapper vulnerabilities", function ()
{
```

Copyright © ***CTDSec*** - All rights reserved

```
  let signers;


  beforeEach(async function () {
    await network.provider.send("hardhat_reset");


    const [deployer, owner1, owner2, automator, multisig, nftOwner, user] = await
ethers.getSigners();
    signers = { deployer, owner1, owner2, automator, multisig, nftOwner, user };
  });


  it("HIGH: non-custodied URM/USDC NFT can receive Fortress liquidity and be
drained by the NFT owner", async function () {
    const env = await deployFortressEnv(1_006_000, signers);
    const usdc = await ethers.getContractAt("HHMockERC20", USDC);


    await usdc.mint(await env.fortress.getAddress(), 10_000_000);
    await env.urm.mint(await env.fortress.getAddress(), 10n * E18);


    await env.fortress.connect(signers.automator).pegDefense();


    expect(await
env.positionManager.ownerOf(NFT_ID_URM)).to.equal(signers.nftOwner.address);
    expect((await usdc.balanceOf(POSITION_MANAGER)) > 0n).to.equal(true);
    expect((await env.urm.balanceOf(POSITION_MANAGER)) > 0n).to.equal(true);
```

Copyright © ***CTDSec*** - All rights reserved

```
    const usdcBefore = await usdc.balanceOf(signers.nftOwner.address);
    const urmBefore = await env.urm.balanceOf(signers.nftOwner.address);
    await
env.positionManager.connect(signers.nftOwner).drainPosition(NFT_ID_URM);
    expect((await usdc.balanceOf(signers.nftOwner.address)) >
usdcBefore).to.equal(true);
    expect((await env.urm.balanceOf(signers.nftOwner.address)) >
urmBefore).to.equal(true);
  });


  it("HIGH: above-peg defense reverts when prior liquidity increase consumes
reserved URM", async function () {
    const env = await deployFortressEnv(1_006_000, signers);
    const usdc = await ethers.getContractAt("HHMockERC20", USDC);
    await env.positionManager.configurePosition(NFT_ID_URM,
signers.owner1.address, USDC, await env.urm.getAddress(), 100, -60, 60, 0);
    await env.fortress.connect(signers.owner1).lockNft(NFT_ID_URM);


    await usdc.mint(await env.fortress.getAddress(), 10_000_000);
    await env.urm.mint(await env.fortress.getAddress(), 50n * E18);


    const Backing = await ethers.getContractFactory("HHMockFortressContract");
    const backing = await Backing.deploy();
    await backing.waitForDeployment();
    await backing.setUsdcStored(100_000_000);
```

Copyright © ***CTDSec*** - All rights reserved

```
    await addFortressContract(env.fortress, await backing.getAddress(), signers);


    const configs = emptyContractConfigs();

    configs[0] = { recipientPercent: 0, syncEnabled: false, backingEnabled: true
};

    await env.fortress.connect(signers.automator).setContractsConfigs(configs);


    await expectRevert(env.fortress.connect(signers.automator).pegDefense(),
"bal");

  });


  it("MEDIUM: syncContract=false still calls fortressPreSync and can block
pegDefense", async function () {

    const env = await deployFortressEnv(1_000_000, signers);

    const SyncContract = await
ethers.getContractFactory("HHMockFortressContract");

    const syncContract = await SyncContract.deploy();

    await syncContract.waitForDeployment();

    await syncContract.setRevertPreSync(true);

    await addFortressContract(env.fortress, await syncContract.getAddress(),
signers);


    const configs = emptyContractConfigs();

    configs[0] = { recipientPercent: 0, syncEnabled: true, backingEnabled: false
};

    await env.fortress.connect(signers.automator).setContractsConfigs(configs);
```

Copyright © ***CTDSec*** - All rights reserved

```
    await network.provider.send("evm_increaseTime", [3601]);
    await network.provider.send("evm_mine");
    await
env.fortress.connect(signers.automator).setConfigs(fortressConfig(false));

    await expectAnyRevert(env.fortress.connect(signers.automator).pegDefense());
  });

  it("HIGH: exact-output RAGE/HESTIA/CIRCLE swaps sell the entire max input
before refunding surplus", async function () {
    const { swapper } = await deploySwapperEnv(signers);
    const user = signers.user;
    const maxInput = 1_000n * E18;
    const exactUrm = 10n * E18;

    for (const [tokenAddress, fnName, poolAddress] of [
      [RAGE, "swapRageToExactUrm", RAGE_USDC_POOL],
      [HESTIA, "swapHestiaToExactUrm", HESTIA_USDC_POOL],
      [CIRCLE, "swapCircleToExactUrm", CIRCLE_WETH_POOL]
    ]) {
      const token = await ethers.getContractAt("HHMockERC20", tokenAddress);
      const pool = await ethers.getContractAt("HHMockPool", poolAddress);
      await token.mint(user.address, maxInput);
```

Copyright © ***CTDSec*** - All rights reserved

```
      await token.connect(user).approve(await swapper.getAddress(), maxInput);

      const tx = await swapper.connect(user)[fnName](exactUrm, maxInput, 500,
(await ethers.provider.getBlock("latest")).timestamp + 1);
      const receipt = await tx.wait();
      expect(receipt.status).to.equal(1);

      const inputSoldFirst = await pool.input0Total();
      expect(inputSoldFirst).to.equal(maxInput);
    }
  });

  it("HIGH: slippage=0 still accepts catastrophic first-leg execution in
exact-output routes", async function () {
    const { swapper } = await deploySwapperEnv(signers);
    const rage = await ethers.getContractAt("HHMockERC20", RAGE);
    const ragePool = await ethers.getContractAt("HHMockPool", RAGE_USDC_POOL);
    await ragePool.setRates(1, E18, E18, E18);

    const maxRage = 1_000n * E18;
    await rage.mint(signers.user.address, maxRage);
    await rage.connect(signers.user).approve(await swapper.getAddress(),
maxRage);

    const latest = await ethers.provider.getBlock("latest");
```

Copyright © ***CTDSec*** - All rights reserved

```
    await swapper.connect(signers.user).swapRageToExactUrm(1000, maxRage, 0,
latest.timestamp + 1);


    expect(await ragePool.input0Total()).to.equal(maxRage);

  });

});
```

Foundry:

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

import {UrmFortress} from "../contracts/UrmFortress.sol";
import {UrmSwapper} from "../contracts/UrmSwapper.sol";
import {IERC20} from "../contracts/IERC20.sol";
import {IUrmSwapper} from "../contracts/IUrmSwapper.sol";
import {INonfungiblePositionManager} from
"../contracts/INonfungiblePositionManager.sol";
import {UrmStructs} from "../contracts/UrmStructs.sol";

interface Vm {
  function etch(address target, bytes calldata newRuntimeBytecode)
external;
  function label(address account, string calldata newLabel) external;
  function prank(address msgSender) external;
  function startPrank(address msgSender) external;
  function stopPrank() external;
  function expectRevert(bytes calldata revertData) external;
  function expectRevert() external;
  function warp(uint256 newTimestamp) external;
}

contract MockERC20 {
  string public name;
  string public symbol;
  uint8 public decimals;
```

Copyright © ***CTDSec*** - All rights reserved

```
  uint256 public totalSupply;
  mapping(address => uint256) public balanceOf;
  mapping(address => mapping(address => uint256)) public allowance;

  function initialize(string calldata name_, string calldata symbol_, uint8
decimals_) external {
    name = name_;
    symbol = symbol_;
    decimals = decimals_;
  }

  function mint(address to, uint256 amount) external {
    balanceOf[to] += amount;
    totalSupply += amount;
  }

  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) {
    require(balanceOf[msg.sender] >= amount, "bal");
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;
    return true;
  }

  function transferFrom(address from, address to, uint256 amount) external
returns (bool) {
    uint256 allowed = allowance[from][msg.sender];
    if (allowed != type(uint256).max) {
      require(allowed >= amount, "allow");
      allowance[from][msg.sender] = allowed - amount;
    }
    require(balanceOf[from] >= amount, "bal");
    balanceOf[from] -= amount;
    balanceOf[to] += amount;
    return true;
  }
}
```

Copyright © ***CTDSec*** - All rights reserved

```
contract MockUrm is MockERC20 {
  uint256 public nextRequestSupplyTime = type(uint256).max;

  function initializeUrm() external {
    this.initialize("URM", "URM", 18);
  }

  function getTotalSupply() external view returns (uint256) {
    return totalSupply;
  }

  function getInitialSupply() external pure returns (uint256) {
    return 0;
  }

  function getMintedSupply() external pure returns (uint256) {
    return 0;
  }

  function getBurnedSupply() external pure returns (uint256) {
    return 0;
  }

  function getNextRequestSupplyTime() external view returns (uint256) {
    return nextRequestSupplyTime;
  }

  function setNextRequestSupplyTime(uint256 value) external {
    nextRequestSupplyTime = value;
  }

  function requestSupply(uint256 percent) external returns (uint256) {
    uint256 amount = (totalSupply * percent) / 100;
    balanceOf[msg.sender] += amount;
    totalSupply += amount;
    return amount;
  }

  function burn(uint256 amount) external {
    require(balanceOf[msg.sender] >= amount, "bal");
    balanceOf[msg.sender] -= amount;
```

```
    totalSupply -= amount;
  }

  function setUrmFortress(address) external {}
  function lockUrmFortress() external {}
}

contract MockFactory {
  address public pool;

  function setPool(address pool_) external {
    pool = pool_;
  }

  function getPool(address, address, uint24) external view returns
(address) {
    return pool;
  }
}

contract MockRce {
  address public owner1;
  address public owner2;
  address public automator;
  address public multisig;
  address public rbp;

  function initialize(address owner1_, address owner2_, address automator_,
address multisig_, address rbp_) external {
    owner1 = owner1_;
    owner2 = owner2_;
    automator = automator_;
    multisig = multisig_;
    rbp = rbp_;
  }

  function getAutomator() external view returns (address) {
    return automator;
  }

  function getOwners() external view returns (address, address) {
    return (owner1, owner2);
```

Copyright © ***CTDSec*** - All rights reserved

```
  }

  function getMultisig() external view returns (address) {
    return multisig;
  }

  function getRageBuyingProtocol() external view returns (address) {
    return rbp;
  }
}

contract MockRbp {
  uint256 public fmv = 1e6;
  uint256 public percentHestia = 50;
  uint256 public percentCircle = 50;

  function getFmv() external view returns (uint256) {
    return fmv;
  }

  function getInvestPercents() external view returns (uint256, uint256) {
    return (percentHestia, percentCircle);
  }

  function getActiveAssetsUsdcValue() external pure returns (uint256) {
    return 0;
  }
}

contract MockOracle {
  uint256 public urmPrice = 1_000_000;
  uint256 public hestiaPrice = 1e6;
  uint256 public circlePrice = 1e6;

  function setUrmPrice(uint256 price) external {
    urmPrice = price;
  }

  function getUrmTwapUsdcPrice(uint256 amount, uint32) external view
returns (uint256) {
    return (amount * urmPrice) / 1e18;
  }
```

Copyright © ***CTDSec*** - All rights reserved

```
  function getWethTwapUsdcPrice(uint256 amount, uint32) external pure
returns (uint256) {
    return (amount * 1e6) / 1e18;
  }

  function getHestiaTwapUsdcPrice(uint256 amount, uint32) external view
returns (uint256) {
    return (amount * hestiaPrice) / 1e18;
  }

  function getCircleTwapUsdcPrice(uint256 amount, uint32) external view
returns (uint256) {
    return (amount * circlePrice) / 1e18;
  }

  function getHestiaCircleTwapUsdcPrice(uint256 hestiaAmount, uint256
circleAmount, uint32) external view returns (uint256) {
    return ((hestiaAmount * hestiaPrice) / 1e18) + ((circleAmount *
circlePrice) / 1e18);
  }

  function getRageTwapUsdcPrice(uint256 amount, uint32) external pure
returns (uint256) {
    return (amount * 1e6) / 1e18;
  }

  function getPrices(uint32) external pure returns (UrmStructs.OraclePrices
memory prices) {
    prices.urmUsdcPrice = 1e6;
    prices.wethTwapUsdcPrice = 1e6;
    prices.hestiaTwapUsdcPrice = 1e6;
    prices.circleTwapUsdcPrice = 1e6;
    prices.rageTwapUsdcPrice = 1e6;
  }
}

contract MockRageOracle {
  function getRageTwapUsdcPrice(uint256 amount, uint32) external pure
returns (uint256) {
    return (amount * 1e6) / 1e18;
  }
```

Copyright © ***CTDSec*** - All rights reserved

```
}

contract MockDepot {
  uint256 public available;

  function setAvailable(uint256 available_) external {
    available = available_;
  }

  function getAvailableFor(address) external view returns (uint256) {
    return available;
  }

  function requestTransfer(address recipient, uint256 amount) external
returns (bool) {
    return
IERC20(0xc0df50143EA93AeC63e38A6ED4E92B378079eA15).transfer(recipient,
amount);
  }

  function selfIncreaseAttribution(uint256 amount) external {

require(IERC20(0xc0df50143EA93AeC63e38A6ED4E92B378079eA15).transferFrom(msg
.sender, address(this), amount), "xfer");
  }
}

contract MockPositionManager {
  struct PositionData {
    address owner;
    address token0;
    address token1;
    uint24 fee;
    int24 tickLower;
    int24 tickUpper;
    uint128 liquidity;
    uint256 added0;
    uint256 added1;
  }

  mapping(uint256 => PositionData) public positionData;
```

Copyright © ***CTDSec*** - All rights reserved

```
  function configurePosition(
    uint256 tokenId,
    address owner,
    address token0,
    address token1,
    uint24 fee,
    int24 tickLower,
    int24 tickUpper,
    uint128 liquidity
  ) external {
    positionData[tokenId] = PositionData({
      owner: owner,
      token0: token0,
      token1: token1,
      fee: fee,
      tickLower: tickLower,
      tickUpper: tickUpper,
      liquidity: liquidity,
      added0: 0,
      added1: 0
    });
  }

  function ownerOf(uint256 tokenId) external view returns (address) {
    return positionData[tokenId].owner;
  }

  function transferFrom(address from, address to, uint256 tokenId) external
{
    require(positionData[tokenId].owner == from, "owner");
    positionData[tokenId].owner = to;
  }

  function positions(uint256 tokenId)
    external
    view
    returns (
      uint96 nonce,
      address operator,
      address token0,
      address token1,
      uint24 fee,
```

Copyright © ***CTDSec*** - All rights reserved

```
      int24 tickLower,
      int24 tickUpper,
      uint128 liquidity,
      uint256 feeGrowthInside0LastX128,
      uint256 feeGrowthInside1LastX128,
      uint128 tokensOwed0,
      uint128 tokensOwed1
    )
  {
    PositionData storage p = positionData[tokenId];
    return (0, address(0), p.token0, p.token1, p.fee, p.tickLower,
p.tickUpper, p.liquidity, 0, 0, 0, 0);
  }

  function
increaseLiquidity(INonfungiblePositionManager.IncreaseLiquidityParams
calldata params)
    external
    returns (uint128 liquidity, uint256 amount0, uint256 amount1)
  {
    PositionData storage p = positionData[params.tokenId];
    amount0 = params.amount0Desired;
    amount1 = params.amount1Desired;
    require(IERC20(p.token0).transferFrom(msg.sender, address(this),
amount0), "xfer0");
    require(IERC20(p.token1).transferFrom(msg.sender, address(this),
amount1), "xfer1");
    p.added0 += amount0;
    p.added1 += amount1;
    liquidity = uint128((amount0 + amount1) / 2);
    p.liquidity += liquidity;
  }

  function collect(INonfungiblePositionManager.CollectParams calldata)
    external
    pure
    returns (uint256 amount0, uint256 amount1)
  {
    return (0, 0);
  }

  function drainPosition(uint256 tokenId) external {
```

```
    PositionData storage p = positionData[tokenId];
    require(msg.sender == p.owner, "owner");
    uint256 amount0 = p.added0;
    uint256 amount1 = p.added1;
    p.added0 = 0;
    p.added1 = 0;
    if (amount0 > 0) require(IERC20(p.token0).transfer(msg.sender,
amount0), "xfer0");
    if (amount1 > 0) require(IERC20(p.token1).transfer(msg.sender,
amount1), "xfer1");
  }
}

contract MockPool {
  uint256 internal constant SCALE = 1e18;
  address public token0;
  address public token1;
  uint160 public sqrtPriceX96;
  uint256 public rate0To1 = SCALE;
  uint256 public rate1To0 = SCALE;
  uint256 public exactOutInputPerOutput0To1 = SCALE;
  uint256 public exactOutInputPerOutput1To0 = SCALE;
  uint256 public input0Total;
  uint256 public input1Total;

  function initialize(address token0_, address token1_) external {
    token0 = token0_;
    token1 = token1_;
    sqrtPriceX96 = 79228162514264337593543950336;
  }

  function setRates(
    uint256 rate0To1_,
    uint256 rate1To0_,
    uint256 exactOutInputPerOutput0To1_,
    uint256 exactOutInputPerOutput1To0_
  ) external {
    rate0To1 = rate0To1_;
    rate1To0 = rate1To0_;
    exactOutInputPerOutput0To1 = exactOutInputPerOutput0To1_;
    exactOutInputPerOutput1To0 = exactOutInputPerOutput1To0_;
  }
```

Copyright © ***CTDSec*** - All rights reserved

```
  function setSqrtPriceX96(uint160 sqrtPriceX96_) external {
    sqrtPriceX96 = sqrtPriceX96_;
  }

  function slot0() external view returns (uint160, int24, uint16, uint16,
uint16, uint8, bool) {
    return (sqrtPriceX96, 0, 0, 0, 0, 0, true);
  }

  function swap(address recipient, bool zeroForOne, int256 amountSpecified,
uint160, bytes calldata data)
    external
    returns (int256 amount0, int256 amount1)
  {
    if (amountSpecified > 0) {
      uint256 amountIn = uint256(amountSpecified);
      uint256 amountOut = zeroForOne ? (amountIn * rate0To1) / SCALE :
(amountIn * rate1To0) / SCALE;
      amount0 = zeroForOne ? int256(amountIn) : -int256(amountOut);
      amount1 = zeroForOne ? -int256(amountOut) : int256(amountIn);
      if (zeroForOne) input0Total += amountIn;
      else input1Total += amountIn;
      IUrmSwapper(msg.sender).uniswapV3SwapCallback(amount0, amount1,
data);
      require(IERC20(zeroForOne ? token1 : token0).transfer(recipient,
amountOut), "out");
    } else {
      uint256 amountOut = uint256(-amountSpecified);
      uint256 amountIn = zeroForOne
        ? (amountOut * exactOutInputPerOutput0To1) / SCALE
        : (amountOut * exactOutInputPerOutput1To0) / SCALE;
      amount0 = zeroForOne ? int256(amountIn) : -int256(amountOut);
      amount1 = zeroForOne ? -int256(amountOut) : int256(amountIn);
      if (zeroForOne) input0Total += amountIn;
      else input1Total += amountIn;
      IUrmSwapper(msg.sender).uniswapV3SwapCallback(amount0, amount1,
data);
      require(IERC20(zeroForOne ? token1 : token0).transfer(recipient,
amountOut), "out");
    }
  }
```

Copyright © ***CTDSec*** - All rights reserved

```
}

contract MockSwapper {
  MockUrm public immutable urm;
  MockERC20 public immutable usdc;

  constructor(MockUrm urm_, MockERC20 usdc_) {
    urm = urm_;
    usdc = usdc_;
  }

  function swapUrmToUsdc(uint256 amount, uint256, uint256) external returns
(uint256) {
    require(urm.transferFrom(msg.sender, address(this), amount), "xfer");
    uint256 usdcOut = amount / 1e12;
    usdc.mint(msg.sender, usdcOut);
    return usdcOut;
  }

  function swapUsdcToUrm(uint256 amount, uint256, uint256) external returns
(uint256) {
    require(usdc.transferFrom(msg.sender, address(this), amount), "xfer");
    uint256 urmOut = amount * 1e12;
    urm.mint(msg.sender, urmOut);
    return urmOut;
  }
}

contract MockFortressContract {
  bool public revertPreSync;
  uint256 public usdcStored;

  function setRevertPreSync(bool value) external {
    revertPreSync = value;
  }

  function setUsdcStored(uint256 value) external {
    usdcStored = value;
  }

  function fortressData() external view returns (uint256, uint256, uint256,
uint256, uint256) {
```

Copyright © ***CTDSec*** - All rights reserved

```
    return (0, usdcStored, 0, 0, 0);
  }

  function fortressPreSync(uint256) external view returns (bool, uint256) {
    require(!revertPreSync, "preSync disabled path called");
    return (false, 0);
  }

  function fortressSync(uint256) external {}
}

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

  address public constant USDC =
0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913;
  address public constant RAGE =
0xc0df50143EA93AeC63e38A6ED4E92B378079eA15;
  address public constant HESTIA =
0xBC7755a153E852CF76cCCDdb4C2e7c368f6259D8;
  address public constant CIRCLE =
0x5baBfc2F240bc5De90Eb7e19D789412dB1dEc402;
  address public constant WETH =
0x4200000000000000000000000000000000000006;
  address public constant RAGE_USDC_POOL =
0xd474B32a5a2BF93453996287D361a00f661E04FF;
  address public constant HESTIA_USDC_POOL =
0x1B39fC4C93EfbE733B8D2770bcfaa46885d5343a;
  address public constant CIRCLE_WETH_POOL =
0xDA679706FF21114AC9faC5198BfF24543F357a16;
  address public constant USDC_WETH_POOL =
0xb2cc224c1c9feE385f8ad6a55b4d94E92359DC59;
  address public constant RAGE_DEPOT =
0x6101c37bB1b3A2A9D8dF14B5a6A97047E7B1628e;
  address public constant RAGE_ORACLE =
0x9B8a45C4A0fBD44158480D9b4B41e0bdCA42874C;
  address public constant RCE = 0x4C15F778Ab59F25D5dFD2dD508236a25eD2813fe;
  address public constant POSITION_MANAGER =
0x03a520b32C04BF3bEEf7BEb72E919cf822Ed34f1;
  address public constant FACTORY =
0x33128a8fC17869897dcE68Ed026d694621f6FDfD;
```

Copyright © ***CTDSec*** - All rights reserved

```
  address internal owner1 = address(0xA11CE);
  address internal owner2 = address(0xB0B);
  address internal automator = address(0xA770);
  address internal multisig = address(0x5151);
  address internal nftOwner = address(0xBADF00D);
  uint256 internal constant NFT_ID_URM = 1;
  uint256 internal constant NFT_ID_RAGE = 4095881;

  struct FortressEnv {
    UrmFortress fortress;
    MockUrm urm;
    MockOracle oracle;
    MockPositionManager positionManager;
    MockPool urmPool;
    MockSwapper swapper;
  }

  function
testHigh_nonCustodiedNftCanReceiveFortressLiquidityAndBeDrainedByExternalNf
tOwner() external {
    FortressEnv memory env = deployFortressEnv(1_006_000);

    MockERC20(USDC).mint(address(env.fortress), 10e6);
    env.urm.mint(address(env.fortress), 10e18);

    vm.prank(automator);
    env.fortress.pegDefense();

    assertTrue(MockPositionManager(POSITION_MANAGER).ownerOf(NFT_ID_URM) ==
nftOwner, "NFT should remain externally owned");
    assertGt(MockERC20(USDC).balanceOf(POSITION_MANAGER), 0, "Fortress USDC
was added to external NFT");
    assertGt(env.urm.balanceOf(POSITION_MANAGER), 0, "Fortress URM was
added to external NFT");

    uint256 ownerUsdcBefore = MockERC20(USDC).balanceOf(nftOwner);
    uint256 ownerUrmBefore = env.urm.balanceOf(nftOwner);
    vm.prank(nftOwner);
    MockPositionManager(POSITION_MANAGER).drainPosition(NFT_ID_URM);

    assertGt(MockERC20(USDC).balanceOf(nftOwner), ownerUsdcBefore,
```

Copyright © ***CTDSec*** - All rights reserved

```
"external NFT owner drains added USDC");
    assertGt(env.urm.balanceOf(nftOwner), ownerUrmBefore, "external NFT
owner drains added URM");
  }

  function
testHigh_abovePegDefenseRevertsWhenPriorLiquidityIncreaseConsumesReservedUr
m() external {
    FortressEnv memory env = deployFortressEnv(1_006_000);
    env.positionManager.configurePosition(NFT_ID_URM, owner1, USDC,
address(env.urm), 100, -60, 60, 0);
    vm.prank(owner1);
    env.fortress.lockNft(NFT_ID_URM);

    MockERC20(USDC).mint(address(env.fortress), 10e6);
    env.urm.mint(address(env.fortress), 50e18);

    MockFortressContract backing = new MockFortressContract();
    backing.setUsdcStored(100e6);
    addFortressContract(env.fortress, address(backing));

    UrmStructs.FortressContractConfig[9] memory configs;
    configs[0] = UrmStructs.FortressContractConfig({
      recipientPercent: 0,
      syncEnabled: false,
      backingEnabled: true
    });
    vm.prank(automator);
    env.fortress.setContractsConfigs(configs);

    vm.expectRevert(bytes("bal"));
    vm.prank(automator);
    env.fortress.pegDefense();
  }

  function
testMedium_syncDisabledStillCallsFortressPreSyncAndCanBlockPegDefense()
external {
    FortressEnv memory env = deployFortressEnv(1_000_000);
    MockFortressContract registered = new MockFortressContract();
    registered.setRevertPreSync(true);
    addFortressContract(env.fortress, address(registered));
```

Copyright © ***CTDSec*** - All rights reserved

```
    UrmStructs.FortressContractConfig[9] memory configs;
    configs[0] = UrmStructs.FortressContractConfig({
      recipientPercent: 0,
      syncEnabled: true,
      backingEnabled: false
    });
    vm.prank(automator);
    env.fortress.setContractsConfigs(configs);

    vm.warp(block.timestamp + 1 hours + 1);
    vm.prank(automator);
    env.fortress.setConfigs(fortressConfig(false));

    vm.expectRevert(bytes("preSync disabled path called"));
    vm.prank(automator);
    env.fortress.pegDefense();
  }

  function
testHigh_rageToExactUrmSellsEntireMaxInputBeforeRefundingSurplus() external
{
    (UrmSwapper swapper, MockUrm urm) = deploySwapperEnv();
    uint256 maxRage = 1_000e18;
    uint256 urmAmount = 10e18;
    address user = address(0xCAFE);

    MockERC20(RAGE).mint(user, maxRage);
    vm.prank(user);
    MockERC20(RAGE).approve(address(swapper), maxRage);

    vm.prank(user);
    (uint256 rageSold, uint256 urmReceived, uint256 rageRefund) =
      swapper.swapRageToExactUrm(urmAmount, maxRage, 500, block.timestamp +
1);

    assertEq(urmReceived, urmAmount, "exact URM output should be
delivered");
    assertGt(rageRefund, 0, "surplus is refunded only after the round
trip");
    assertLt(rageSold, maxRage, "net return masks that all max input was
sold first");
```

Copyright © **_CTDSec_** - All rights reserved

```
    assertEq(MockPool(RAGE_USDC_POOL).input0Total(), maxRage, "first leg
sells the entire maxRage");
    assertEq(urm.balanceOf(user), urmAmount, "user receives exact URM");
  }

  function
testHigh_hestiaToExactUrmSellsEntireMaxInputBeforeRefundingSurplus()
external {
    (UrmSwapper swapper,) = deploySwapperEnv();
    uint256 maxHestia = 1_000e18;
    uint256 urmAmount = 10e18;
    address user = address(0xCAFE);

    MockERC20(HESTIA).mint(user, maxHestia);
    vm.prank(user);
    MockERC20(HESTIA).approve(address(swapper), maxHestia);

    vm.prank(user);
    (uint256 hestiaSold,, uint256 hestiaRefund) =
      swapper.swapHestiaToExactUrm(urmAmount, maxHestia, 500,
block.timestamp + 1);

    assertGt(hestiaRefund, 0, "surplus is refunded only after the round
trip");
    assertLt(hestiaSold, maxHestia, "net return masks that all max input
was sold first");
    assertEq(MockPool(HESTIA_USDC_POOL).input0Total(), maxHestia, "first
leg sells the entire maxHestia");
  }

  function
testHigh_circleToExactUrmSellsEntireMaxInputBeforeRefundingSurplus()
external {
    (UrmSwapper swapper,) = deploySwapperEnv();
    uint256 maxCircle = 1_000e18;
    uint256 urmAmount = 10e18;
    address user = address(0xCAFE);

    MockERC20(CIRCLE).mint(user, maxCircle);
    vm.prank(user);
    MockERC20(CIRCLE).approve(address(swapper), maxCircle);
```

Copyright © ***CTDSec*** - All rights reserved

```
    vm.prank(user);
    (uint256 circleSold,, uint256 circleRefund) =
      swapper.swapCircleToExactUrm(urmAmount, maxCircle, 500,
block.timestamp + 1);

    assertGt(circleRefund, 0, "surplus is refunded only after the round
trip");
    assertLt(circleSold, maxCircle, "net return masks that all max input
was sold first");
    assertEq(MockPool(CIRCLE_WETH_POOL).input0Total(), maxCircle, "first
leg sells the entire maxCircle");
  }

  function
testHigh_zeroSlippageOnExactOutputInputLegStillAcceptsCatastrophicExecution
() external {
    (UrmSwapper swapper,) = deploySwapperEnv();
    MockPool(RAGE_USDC_POOL).setRates(1, 1e18, 1e18, 1e18);

    uint256 maxRage = 1_000e18;
    uint256 tinyUrmOutput = 1_000;
    address user = address(0xCAFE);

    MockERC20(RAGE).mint(user, maxRage);
    vm.prank(user);
    MockERC20(RAGE).approve(address(swapper), maxRage);

    vm.prank(user);
    (uint256 rageSold, uint256 urmReceived, uint256 rageRefund) =
      swapper.swapRageToExactUrm(tinyUrmOutput, maxRage, 0, block.timestamp
+ 1);

    assertEq(urmReceived, tinyUrmOutput, "exact output is technically
satisfied");
    assertEq(rageRefund, 0, "no refund remains after the bad first-leg
execution");
    assertEq(rageSold, maxRage, "slippage=0 did not prevent selling all
input at a terrible rate");
  }

  function deployFortressEnv(uint256 urmPrice) internal returns
(FortressEnv memory env) {
```

Copyright © ***CTDSec*** - All rights reserved

```
    installCommonAddressMocks();

    env.urm = new MockUrm();
    env.urm.initializeUrm();
    env.oracle = new MockOracle();
    env.oracle.setUrmPrice(urmPrice);
    env.swapper = new MockSwapper(env.urm, MockERC20(USDC));

    address urmPoolAddress = address(0x1001001);
    MockPool poolImpl = new MockPool();
    vm.etch(urmPoolAddress, address(poolImpl).code);
    env.urmPool = MockPool(urmPoolAddress);
    env.urmPool.initialize(USDC, address(env.urm));

    MockFactory(FACTORY).setPool(urmPoolAddress);
    env.positionManager = MockPositionManager(POSITION_MANAGER);
    env.positionManager.configurePosition(NFT_ID_URM, nftOwner, USDC,
address(env.urm), 100, -60, 60, 0);
    env.positionManager.configurePosition(NFT_ID_RAGE, address(0x1234),
RAGE, USDC, 100, -60, 60, 0);

    env.fortress = new UrmFortress(address(env.urm), address(env.swapper),
address(env.oracle), urmPoolAddress, NFT_ID_URM);
  }

  function deploySwapperEnv() internal returns (UrmSwapper swapper, MockUrm
urm) {
    installToken(USDC, "USDC", "USDC", 18);
    installToken(RAGE, "RAGE", "RAGE", 18);
    installToken(HESTIA, "HESTIA", "HESTIA", 18);
    installToken(CIRCLE, "CIRCLE", "CIRCLE", 18);
    installToken(WETH, "WETH", "WETH", 18);

    urm = new MockUrm();
    urm.initializeUrm();

    address urmPool = address(0x2002002);
    installPool(urmPool, USDC, address(urm));
    installPool(RAGE_USDC_POOL, RAGE, USDC);
    installPool(HESTIA_USDC_POOL, HESTIA, USDC);
    installPool(CIRCLE_WETH_POOL, CIRCLE, WETH);
    installPool(USDC_WETH_POOL, WETH, USDC);
```

Copyright © ***CTDSec*** - All rights reserved

```
    MockPool(urmPool).setRates(1e18, 1e18, 1e18, 1e18);
    MockPool(RAGE_USDC_POOL).setRates(1e18, 1e18, 1e18, 1e18);
    MockPool(HESTIA_USDC_POOL).setRates(1e18, 1e18, 1e18, 1e18);
    MockPool(CIRCLE_WETH_POOL).setRates(1e18, 1e18, 1e18, 1e18);
    MockPool(USDC_WETH_POOL).setRates(1e18, 1e18, 1e18, 1e18);

    MockERC20(USDC).mint(RAGE_USDC_POOL, 10_000e18);
    MockERC20(RAGE).mint(RAGE_USDC_POOL, 10_000e18);
    MockERC20(USDC).mint(HESTIA_USDC_POOL, 10_000e18);
    MockERC20(HESTIA).mint(HESTIA_USDC_POOL, 10_000e18);
    MockERC20(WETH).mint(CIRCLE_WETH_POOL, 10_000e18);
    MockERC20(CIRCLE).mint(CIRCLE_WETH_POOL, 10_000e18);
    MockERC20(USDC).mint(USDC_WETH_POOL, 10_000e18);
    MockERC20(WETH).mint(USDC_WETH_POOL, 10_000e18);
    urm.mint(urmPool, 10_000e18);

    MockRce rceImpl = new MockRce();
    MockRbp rbpImpl = new MockRbp();
    vm.etch(RCE, address(rceImpl).code);
    vm.etch(address(0xff70Cd1E1931372F869c936582a7F42e49B6DA4c),
address(rbpImpl).code);
    MockRce(RCE).initialize(owner1, owner2, automator, multisig,
address(0xff70Cd1E1931372F869c936582a7F42e49B6DA4c));

    swapper = new UrmSwapper(address(urm), urmPool, address(new
MockOracle()));
  }

  function installCommonAddressMocks() internal {
    installToken(USDC, "USDC", "USDC", 6);
    installToken(RAGE, "RAGE", "RAGE", 18);
    installToken(HESTIA, "HESTIA", "HESTIA", 18);
    installToken(CIRCLE, "CIRCLE", "CIRCLE", 18);
    installPool(RAGE_USDC_POOL, RAGE, USDC);

    MockDepot depotImpl = new MockDepot();
    MockRageOracle rageOracleImpl = new MockRageOracle();
    MockRce rceImpl = new MockRce();
    MockFactory factoryImpl = new MockFactory();
    MockPositionManager positionManagerImpl = new MockPositionManager();
    MockRbp rbpImpl = new MockRbp();
```

Copyright © ***CTDSec*** - All rights reserved

```
    vm.etch(RAGE_DEPOT, address(depotImpl).code);
    vm.etch(RAGE_ORACLE, address(rageOracleImpl).code);
    vm.etch(RCE, address(rceImpl).code);
    vm.etch(FACTORY, address(factoryImpl).code);
    vm.etch(POSITION_MANAGER, address(positionManagerImpl).code);
    vm.etch(address(0xff70Cd1E1931372F869c936582a7F42e49B6DA4c),
address(rbpImpl).code);
    MockRce(RCE).initialize(owner1, owner2, automator, multisig,
address(0xff70Cd1E1931372F869c936582a7F42e49B6DA4c));
  }

  function installToken(address target, string memory name, string memory
symbol, uint8 decimals) internal {
    MockERC20 tokenImpl = new MockERC20();
    vm.etch(target, address(tokenImpl).code);
    MockERC20(target).initialize(name, symbol, decimals);
  }

  function installPool(address target, address token0, address token1)
internal {
    MockPool poolImpl = new MockPool();
    vm.etch(target, address(poolImpl).code);
    MockPool(target).initialize(token0, token1);
  }

  function addFortressContract(UrmFortress fortress, address contractAddr)
internal {
    vm.prank(owner1);
    fortress.addContract(contractAddr, 1);
    vm.prank(owner2);
    fortress.addContract(contractAddr, 1);
  }

  function fortressConfig(bool syncContract) internal pure returns
(UrmStructs.FortressConfig memory) {
    return UrmStructs.FortressConfig({
      defenseSize: 50e6,
      assetsValuePercent: 75,
      ragePoolPercent: 50,
      twap: 180,
      slippage: 500,
```

Copyright © ***CTDSec*** - All rights reserved

```
      abovePegThreshold: 1_005_000,
      belowPegThreshold: 999_000,
      belowPegThreshold2: 995_000,
      abovePegDelay: 2 minutes,
      belowPegDelay: 2 minutes,
      abovePegBuy: 50,
      pegDefenseTarget: 0,
      requestSupplyPercent: 10,
      requestSupplyMin: 50_000e18,
      burnSupplyMin: 100_000e18,
      pegDefensePublic: false,
      syncContract: syncContract,
      collectFeesDelay: 7 days,
      liquidityIncrease: true
    });
  }

  function assertTrue(bool condition, string memory message) internal pure
{
    require(condition, message);
  }

  function assertEq(uint256 actual, uint256 expected, string memory
message) internal pure {
    require(actual == expected, message);
  }

  function assertGt(uint256 actual, uint256 min, string memory message)
internal pure {
    require(actual > min, message);
  }

  function assertLt(uint256 actual, uint256 max, string memory message)
internal pure {
    require(actual < max, message);
  }
}
```

Copyright © ***CTDSec*** - All rights reserved

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 and no unresolved critical issues remain.

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

Copyright © ***CTDSec*** - All rights reserved