code4rena

Morpheus

Smart Contract Security Assessment

Audit dates: Aug 15 — Aug 25, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Morpheus smart contract system. The audit took place from August 15 to August 25, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 26 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Morpheus team.

Scope

The code under review can be found within the C4 Morpheus repository, and is composed of 6 smart contracts written in the Solidity programming language and includes 390 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

Medium Risk Findings (4)

[M-01] Same heartbeat for multiple price feeds is vulnerable

Submitted by newspacexyz, also found by 0xDemon, 0xgh0st, 0xVI, boredpukar, dd0x7e8, dimah7, dimulski, honey-k12, hrmneffdii, ironsidesec, itsravin0x, KlosMitSoss, natachi, Olugbenga, oxchsyston, queen, rashmor, slowbugmayor, vesko210, Yaneca_b, ZanyBonzy, and ZeroEx

https://github.com/code-423n4/2025-08-morpheus/blob/d275e9035b7e703efe26cdee6543e78f452774bb/contracts/capital-protocol/ChainLinkDataConsumer.sol#L91

Finding description and impact

`ChainLinkDataConsumer` uses the same heartbeat `allowedPriceUpdateDelay` for multiple feeds when checking if the data feed is fresh. The issue with this is that the USDC/USD oracle has a 24 hour heartbeat, whereas the average has a heartbeat of 1 hour. Since they use the same heartbeat, the heartbeat needs to be slower of the two or else the contract would be nonfunctional most of the time. The issue is that it would allow the consumption of potentially very stale data from the non-USDC feed.

Impact is that either near constant downtime or insufficient staleness checks will happen.

Recommended Mitigation Steps

Use individual heartbeat periods

ref: code423n4 mentioned this on Aug 5, 2023

Using the same heartbeat for all Chainlink feeds will either result in frequent reverts or stale prices: code-423n4/2023-07-tapioca-findings#1505

[M-02] Inconsistent balance accounting in stETH deposits leads to DOS of core functions and reward loss

Submitted by KlosMitSoss, also found by 0xDemon, Aristos, Avalance, Ivcho332, mishoko, mrMorningstar, newspacexyz, Olugbenga, SOPROBRO, Utsav, and Yaneca_b https://github.com/code-423n4/2025-08-morpheus/blob/d275e9035b7e703efe26cdee6543e78f452774bb/contracts/capital-protocol/Distributor.sol#L295-L301

https://github.com/code-423n4/2025-08-morpheus/blob/d275e9035b7e703efe26cdee6543e78f452774bb/contracts/capital-protocol/Distributor.sol#L384-L386

Finding Description and Impact

When stETH is staked, the `DepositPool::_stake()` function only stores the actual balance difference, which might be different from the amount that was provided for the transfer due to rounding down:

```
// https://docs.lido.fi/guides/lido-tokens-integration-guide/#steth-internals-share-mechanics
uint256 balanceBefore_ = IERC20(depositToken).balanceOf(address(this));
IERC20(depositToken).safeTransferFrom(_msgSender(), address(this), amount_);
uint256 balanceAfter_ = IERC20(depositToken).balanceOf(address(this));

amount_ = balanceAfter_ - balanceBefore_;
```

https://github.com/code-423n4/2025-08-morpheus/blob/a65c254e4c3133c32c05b80bf2bd6ff9eced68e2/contracts/capital-protocol/DepositPool.sol#L379-L384

The link from the Lido documentation provided in the code comment explains that the "stETH balance calculation includes integer division, and there is a common case when the whole stETH balance can't be transferred from the account while leaving the last 1-2 wei on the sender's account. The same thing can actually happen at any transfer or deposit transaction. In the future, when the stETH/share rate becomes greater, the error can become a bit bigger. To avoid it, one can use transferShares to be precise."

This means that the actual transferred amount could be lower than the amount that the transfer operation was called with due to rounding down. Hence, the actual balance difference is used in `DepositPool::_stake()`. This mechanism is also documented in the Morpheus documentation.

However, when `Distributor::supply()` is called within `DepositPool::_stake()`, which transfers the `amount_` it was called with from the deposit pool to the distributor, the `amount_` is stored in `depositPool.deposited` and `depositPool.lastUnderlyingBalance` instead of the actual balance difference:

```
IERC20(depositPool.token).safeTransferFrom(depositPoolAddress_,
address(this), amount_);

depositPool.deposited += amount_;
depositPool.lastUnderlyingBalance += amount_;
```

https://github.com/code-423n4/2025-08-morpheus/blob/a65c254e4c3133c32c05b80bf2bd6ff9eced68e2/contracts/capital-protocol/Distributor.sol#L295-L301

As a result, the value stored in `depositPool.lastUnderlyingBalance` and `depositPool.deposited` could be larger than the actual balance of the contract. This leads to an issue in `Distributor::distributeRewards()` where this difference causes an underflow:

```
uint256 balance_ = IERC20(yieldToken_).balanceOf(address(this));
uint256 decimals_ = IERC20Metadata(yieldToken_).decimals();
uint256 underlyingYield_ = (balance_ -
depositPool.lastUnderlyingBalance).to18(decimals_);
```

https://github.com/code-423n4/2025-08-morpheus/blob/a65c254e4c3133c32c05b80bf2bd6ff9eced68e2/contracts/capital-protocol/Distributor.sol#L384-L386

When the strategy of the deposit pool is `Strategy.NONE` (which is the case for stETH according to this code comment in `IDistributor`), the `yieldToken_` will be the `depositToken`.

As long as the yield that stETH accrued does not exceed this difference between the stored value and actual balance, this leads to a DOS of `Distributor::distributeRewards()` and all functions that call it, including stake(), withdraw(), lockClaim(), and claim functions.

When stETH has accrued enough yield, this DOS will be resolved. However, the yield needed to balance out the values will not be recognized as actual yield. As a result, it will not be included in the rewards calculation. This leads to a loss of rewards for the stakers of the stETH deposit pool.

This issue also exists during the migration flow, where `Distributor::supply()` is called:

```
uint256 remainder_ = IERC20(depositToken).balanceOf(address(this)) -
totalDepositedInPublicPools;
require(remainder_ > 0, "DS: yield for token is zero");
IERC20(depositToken).transfer(distributor, remainder_);
```

```
IDistributor(distributor).supply(rewardPoolIndex_,
totalDepositedInPublicPools);

isMigrationOver = true;
```

https://github.com/code-423n4/2025-08-morpheus/blob/a65c254e4c3133c32c05b80bf2bd6ff9eced68e2/contracts/capital-protocol/DepositPool.sol#L151-L157

According to this documentation, `migrate()` will be used for stETH. Furthermore, the `DepositPool.test.ts` also uses a deposit pool with stETH as the depositToken and Strategy.NONE as its strategy.

Recommended Mitigation Steps

Only store the balance difference every time stETH is transferred, similar to the approach used in `DepositPool::_stake()`.

Proof of Concept

Add the test to `POC.test.ts` and run `npx hardhat test --grep "should expose lastUnderlyingBalance tracking issue due to stETH rounding during stake"`

```
  describe('#stETH rounding behavior - lastUnderlyingBalance vs actual
balance', () => {
    let stETHDepositPool: DepositPool;

    beforeEach(async () => {
      // Create a deposit pool with stETH as the deposit token
      const lib1 = await (await
ethers.getContractFactory('ReferrerLib')).deploy();
      const lib2 = await (await
ethers.getContractFactory('LockMultiplierMath')).deploy();
      const stETHDepositPoolFactory = await
ethers.getContractFactory('DepositPool', {
        libraries: {
          ReferrerLib: await lib1.getAddress(),
          LockMultiplierMath: await lib2.getAddress(),
        },
      });
      const stETHDepositPoolImpl = await
stETHDepositPoolFactory.deploy();
      const proxyFactory = await
ethers.getContractFactory('ERC1967Proxy');
```

```
      const stETHDepositPoolProxy = await proxyFactory.deploy(await
stETHDepositPoolImpl.getAddress(), '0x');
      stETHDepositPool = stETHDepositPoolFactory.attach(await
stETHDepositPoolProxy.getAddress()) as DepositPool;

      await stETHDepositPool.DepositPool_init(await stETH.getAddress(),
await distributor.getAddress());

      // Add stETH deposit pool to distributor
      await distributor.addDepositPool(
        publicRewardPoolId,
        await stETHDepositPool.getAddress(),
        await stETH.getAddress(),
        'ETH/USD',
        Strategy.NONE,
      );

      // Complete migration
      await stETHDepositPool.migrate(publicRewardPoolId);

      // Setup reward pool timestamp (required before any staking)
      await
distributor.setRewardPoolLastCalculatedTimestamp(publicRewardPoolId, 1);

      // Set up a realistic total pooled ether amount to trigger rounding
      await stETH.setTotalPooledEther(wei('123456.789123456789'));

      // Set time to start reward distribution
      await setNextTime(oneDay * 11);
    });

    it('should expose lastUnderlyingBalance tracking issue due to stETH
rounding during stake', async () => {
      const testAmounts = [
        wei('1.000000001'),      // Small fractional amount
      ];

      for (const amount of testAmounts) {
        // Mint exact stETH amount to alice
        await stETH.mint(alice.address, amount);
        await stETH.connect(alice).approve(await
stETHDepositPool.getAddress(), amount);

        // Get initial balances
        const distributorInitialBalance = await stETH.balanceOf(await
distributor.getAddress());
```

```
        // Get initial distributor pool state
        const initialDepositPoolData = await
distributor.depositPools(publicRewardPoolId, await
stETHDepositPool.getAddress());
        const initialLastUnderlyingBalance =
initialDepositPoolData.lastUnderlyingBalance;

        // Perform stake operation - this triggers supply() in
distributor
        // The distributor.supply() will update lastUnderlyingBalance
with the amount parameter
        // But the actual stETH transfer may round down due to stETH
mechanics
        await stETHDepositPool.connect(alice).stake(publicRewardPoolId,
amount, 0, ZERO_ADDR);

        // Get final balances
        const distributorFinalBalance = await stETH.balanceOf(await
distributor.getAddress());

expect(distributorFinalBalance).to.be.gt(distributorInitialBalance);

        // Get final distributor pool state
        const finalDepositPoolData = await
distributor.depositPools(publicRewardPoolId, await
stETHDepositPool.getAddress());
        const finalLastUnderlyingBalance =
finalDepositPoolData.lastUnderlyingBalance;

expect(finalLastUnderlyingBalance).to.be.gt(initialLastUnderlyingBalance)
;

        // lastUnderlyingBalance > actual transferred stETH (distributor
balance)

expect(finalLastUnderlyingBalance).to.be.gt(distributorFinalBalance);

        // distributeRewards reverts due to 0x11 which is the panic code
for underflow
        await expect(
          distributor.distributeRewards(publicRewardPoolId)
        ).to.be.revertedWithPanic(0x11);

        // the same goes for the functions where this is called (like
stake and withdraw):
        // Mint exact stETH amount to alice
        await stETH.mint(alice.address, amount);
        await stETH.connect(alice).approve(await
stETHDepositPool.getAddress(), amount);

        await expect(
          stETHDepositPool.connect(alice).stake(publicRewardPoolId,
amount, 0, ZERO_ADDR)
        ).to.be.revertedWithPanic(0x11);
      }
    });
  });
```

[M-03] Protocol doesn't properly handle Aave Pool changes

Submitted by ZanyBonzy, also found by anchabadze

https://github.com/code-423n4/2025-08-morpheus/blob/d275e9035b7e703efe26cdee6543e78f452774bb/contracts/capital-protocol/Distributor.sol#L135-L141

https://github.com/code-423n4/2025-08-morpheus/blob/d275e9035b7e703efe26cdee6543e78f452774bb/contracts/capital-protocol/Distributor.sol#L226-L237

https://github.com/code-423n4/2025-08-morpheus/blob/d275e9035b7e703efe26cdee6543e78f452774bb/contracts/capital-protocol/Distributor.sol#L285-L328

Finding description and impact

`setAavePool` is used to set new aave pool address, presumably to handle pool changes, upgrades, or migration situations.

```
    function setAavePool(address value_) public onlyOwner {
        require(value_ != address(0), "DR: invalid Aave pool address");

        aavePool = value_;//<<===

        emit AavePoolSet(value_);
    }
```

There are two issues with this however.

1. Aave recommends executing calls to pools dynamically, i.e querying `PoolAddressProvider` for the current pool address. The protocol doesn't do this, opting to allow the owner to set the pool address whenever it is required, leaving a window between which the pool in use will be deprecated, leading to DOS issues until a new pool is set.
2. The more serious issue, is the missing approval revocation for old pool and granting for the new one. This is because max token approvals are only set once when deposit pools are added in `addDepositPool()`. After updating aavePool, `supply`, `withdraw`, and `withdrawYield` calls for existing Aave-strategy deposit pools fail due to missing approvals for the new pool. The `isDepositTokenAdded` mapping prevents re-adding the same token to re-approve, locking funds or halting deposits.

As a result, the protocol is stuck in a state where, the current aave pool is deprecated and unusable, and the new pool that is set cannot be used due to missing approvals for the new pool, locking down operations.

```
        address aToken_ = address(0);
        if (strategy_ == Strategy.AAVE) {
            (aToken_, , ) =
AaveIPoolDataProvider(aavePoolDataProvider).getReserveTokensAddresses(tok
en_);

            IERC20(token_).safeApprove(aavePool,
type(uint256).max);//<<===
            IERC20(aToken_).approve(aavePool, type(uint256).max);//<<===
        }

        DepositPool memory depositPool_ = DepositPool(token_,
chainLinkPath_, 0, 0, 0, strategy_, aToken_, true);
```

Aave pools have been designed to be updated or migrated since V1, and major pool addresses have been updated in the past (e.g., during major version upgrades like V2 to V3 or pool migrations), so this is a realistic scenario, especially over the long term.

Recommended Mitigation Steps

Set immutable `poolAddressProvider` instead and query it everytime `aavePool` functions need to be accessed. Approve the queried address to spend the needed amount of tokens for each transaction.

Proof of Concept

We add the test below to Distributor.test.ts

```
  describe('Bug - setAavePool approval issue', () => {
    const maxUint256 = 2n**256n - 1n;
    beforeEach(async () => {
      await chainLinkDataConsumerMock.setAnswer(dp1Info.chainLinkPath,
wei(1));
      await distributor.addDepositPool(
        dp1Info.rewardPoolId,
        dp1Info.depositPool,
        dp1Info.depositToken,
        dp1Info.chainLinkPath,
        dp1Info.strategy,
      );
      await dp1Info.depositToken.mint(BOB, wei(1000));
      await
dp1Info.depositToken.connect(BOB).approve(dp1Info.depositPool,
wei(1000));
      await
distributor.setRewardPoolLastCalculatedTimestamp(publicRewardPoolId, 1);
    });

    it('should fail supply after changing aavePool due to missing
approvals', async () => {
      // Initial supply works with original aavePool
      await dp1Info.depositPool.connect(BOB).supply(publicRewardPoolId,
wei(100));
      let dp1 = await distributor.depositPools(publicRewardPoolId,
dp1Info.depositPool);
      expect(dp1.deposited).to.eq(wei(100));
      expect(await
dp1Info.aToken.balanceOf(distributor)).to.eq(wei(100));

      // Change aavePool
      const newAavePool = await
deployAavePoolMock(aavePoolDataProviderMock);
      await distributor.setAavePool(newAavePool);

      // Attempt to supply again, should fail due to missing approval
      await expect(
        dp1Info.depositPool.connect(BOB).supply(publicRewardPoolId,
wei(50))
      ).to.be.revertedWith('ERC20: insufficient allowance');

      // Verify no new deposits
      dp1 = await distributor.depositPools(publicRewardPoolId,
dp1Info.depositPool);
      expect(dp1.deposited).to.eq(wei(100));
      expect(await
dp1Info.aToken.balanceOf(distributor)).to.eq(wei(100));

      // Attempt to withdraw, should also fail
      await expect(
        dp1Info.depositPool.connect(BOB).withdraw(publicRewardPoolId,
wei(50))
      ).to.be.revertedWith('ERC20: insufficient allowance');
    });

    it('should show old pool still has approvals while new pool has
none', async () => {
      // Check that original Aave pool has approvals
      const originalAllowance = await
dp1Info.depositToken.allowance(distributor, aavePoolMock);
      expect(originalAllowance).to.eq(maxUint256);

      // Update to new Aave pool
      const newAavePool = await
deployAavePoolMock(aavePoolDataProviderMock);
      await distributor.setAavePool(newAavePool);

      // New pool has no approvals
      const newAllowance = await
dp1Info.depositToken.allowance(distributor, newAavePool);
      expect(newAllowance).to.eq(0);

      // Old pool still has approvals (potential security concern)
      const oldAllowance = await
dp1Info.depositToken.allowance(distributor, aavePoolMock);
      expect(oldAllowance).to.eq(maxUint256);
    });
  });
```

[M-04] Yield withdrawal blocked by zero reward early return

Submitted by KlosMitSoss

https://github.com/code-423n4/2025-08-morpheus/blob/a65c254e4c3133c32c05b80bf2bd6ff9eced68e2/contracts/capital-protocol/Distributor.sol#L346

https://github.com/code-423n4/2025-08-morpheus/blob/a65c254e4c3133c32c05b80bf2bd6ff9eced68e2/contracts/capital-protocol/Distributor.sol#L482-L483

Finding Description and Impact

When withdrawing yield, it is calculated as the difference between `lastUnderlyingBalance` and `deposited` for the given `depositPool`. The `lastUnderlyingBalance` is updated in `distributeRewards()` to reflect the current token balance, which includes accumulated yield. However, this function returns early when the calculated `rewards_` amount for the period between `lastCalculatedTimestamp_` and `block.timestamp` is zero:

```
uint256 rewards_ = IRewardPool(rewardPool).getPeriodRewards(
    rewardPoolIndex_,
    lastCalculatedTimestamp_,
    uint128(block.timestamp)
);
if (rewards_ == 0) return;
```

https://github.com/code-423n4/2025-08-morpheus/blob/a65c254e4c3133c32c05b80bf2bd6ff9eced68e2/contracts/capital-protocol/Distributor.sol#L340-L346

This prevents `lastUnderlyingBalance` from being updated when `rewards_` is zero, causing the accounting system to fail to recognize newly accumulated yield and making it impossible to withdraw.

Rewards are only calculated up to the maxEndTime, which represents when reward distribution for a given pool ends completely. After this point, no more rewards are distributed from that pool.

When `getPeriodRewards()` returns zero for any period after `maxEndTime`, it will permanently return zero for all future periods. This means `distributeRewards()` will always return early without updating `lastUnderlyingBalance`, creating a permanent lock on any yield that accumulated since `lastCalculatedTimestamp_` and any yield that accumulates after the reward distribution period ends.

The problem is particularly severe because:

1. Users can remain staked after `maxEndTime` and their tokens continue generating yield
2. The yield accounting becomes permanently broken once rewards end
3. All yield generated after `maxEndTime` becomes permanently inaccessible
4. This affects the entire deposit pool, not just individual users

If `lastCalculatedTimestamp_ == maxEndTime - 100`, `getPeriodRewards(rewardPoolIndex_, lastCalculatedTimestamp_, maxEndTime) == 0`, and `yield == 10e18` during that time, this yield amount will be permanently stuck. Any continuing yield accumulation after `maxEndTime` will also become unwithdrawable.

Recommended Mitigation Steps

Implement functionality that allows withdrawing yield even when no rewards accrue. For example:

1. Add a mechanism to update `lastUnderlyingBalance` independent of reward calculations
2. Implement a fallback method for yield withdrawal after reward distribution ends

Proof of Concept

Add the following file to `./test/` and run `npx hardhat test --grep "should show yield stuck"`

```
import { SignerWithAddress } from '@nomicfoundation/hardhat-
ethers/signers';
import { expect } from 'chai';
import { ethers } from 'hardhat';

import {
  AavePoolDataProviderMock,
  AavePoolMock,
  ArbitrumBridgeGatewayRouterMock, // Mock contracts
  ChainLinkAggregatorV3Mock,
  ChainLinkDataConsumer,
  DepositPool,
  Distributor,
  ERC20Token,
  IDepositPool,
  IL1SenderV2,
  IL2TokenReceiverV2, // Interfaces
  IRewardPool,
  L1SenderV2,
  L2MessageReceiver,
  L2TokenReceiverV2,
  LayerZeroEndpointV2Mock,
  MOR,
  MOROFT,
  NonfungiblePositionManagerMock,
  RewardPool,
  StETHMock,
  UniswapSwapRouterMock,
  WStETHMock,
} from '@/generated-types/ethers';
import { PRECISION, ZERO_ADDR } from '@/scripts/utils/constants';
import { wei } from '@/scripts/utils/utils';
import { getCurrentBlockTime, setNextTime, setTime } from
'@/test/helpers/block-helper';
import { Reverter } from '@/test/helpers/reverter';

describe('Morpheus Capital Protocol - POC Test Suite', function () {
  // Signers
  let owner: SignerWithAddress;
  let alice: SignerWithAddress;
  let bob: SignerWithAddress;
  let charlie: SignerWithAddress;
  let treasury: SignerWithAddress;
  let referrer1: SignerWithAddress;
  let referrer2: SignerWithAddress;

  // Core Capital Protocol Contracts
  let chainLinkDataConsumer: ChainLinkDataConsumer;
  let depositPool: DepositPool;
  let distributor: Distributor;
  let l1Sender: L1SenderV2;
  let l2TokenReceiver: L2TokenReceiverV2;
  let rewardPool: RewardPool;

  // Mock Contracts and Dependencies
  let aavePool: AavePoolMock;
  let aavePoolDataProvider: AavePoolDataProviderMock;
  let stETH: StETHMock;
  let wstETH: WStETHMock;
  let depositToken: ERC20Token;
  let rewardToken: ERC20Token;
  let mor: MOR;
  let morOft: MOROFT;
  let swapRouter: UniswapSwapRouterMock;
  let nonfungiblePositionManager: NonfungiblePositionManagerMock;
  let arbitrumBridge: ArbitrumBridgeGatewayRouterMock;
  let lzEndpointL1: LayerZeroEndpointV2Mock;
  let lzEndpointL2: LayerZeroEndpointV2Mock;
  let l2MessageReceiver: L2MessageReceiver;

  // Price Feed Mocks
  let ethUsdFeed: ChainLinkAggregatorV3Mock;
  let btcUsdFeed: ChainLinkAggregatorV3Mock;
  let wbtcBtcFeed: ChainLinkAggregatorV3Mock;

  // Test Constants
  const oneDay = 86400;
  const oneHour = 3600;
  const publicRewardPoolId = 0;
  const privateRewardPoolId = 1;
  const l1ChainId = 101;
  const l2ChainId = 110;

  // Reverter for test isolation
  const reverter = new Reverter();

  // Enums
  enum Strategy {
    NONE, // Direct deposit, no yield
    NO_YIELD, // No yield strategy (private pools)
    AAVE, // Aave yield strategy
  }

  // Helper Functions
  async function deployERC20Token(): Promise<ERC20Token> {
    const factory = await ethers.getContractFactory('ERC20Token');
    return await factory.deploy();
  }

  async function deployMocks() {
    // Deploy price feed mocks
    const aggregatorFactory = await
ethers.getContractFactory('ChainLinkAggregatorV3Mock');
    ethUsdFeed = await aggregatorFactory.deploy(18);
    btcUsdFeed = await aggregatorFactory.deploy(8);
    wbtcBtcFeed = await aggregatorFactory.deploy(8);

    // Set initial prices
    await ethUsdFeed.setAnswerResult(wei(2000, 18)); // $2000/ETH
    await btcUsdFeed.setAnswerResult(wei(40000, 8)); // $40000/BTC
    await wbtcBtcFeed.setAnswerResult(wei(1, 8)); // 1:1 WBTC/BTC

    // Deploy Aave mocks
    const aaveDataProviderFactory = await
ethers.getContractFactory('AavePoolDataProviderMock');
    aavePoolDataProvider = await aaveDataProviderFactory.deploy();

    const aavePoolFactory = await
ethers.getContractFactory('AavePoolMock');
    aavePool = await aavePoolFactory.deploy(aavePoolDataProvider);

    // Deploy token mocks
    const stETHFactory = await ethers.getContractFactory('StETHMock');
    stETH = await stETHFactory.deploy();
    const wstETHFactory = await ethers.getContractFactory('WStETHMock');
    wstETH = await wstETHFactory.deploy(stETH);

    depositToken = await deployERC20Token();
    rewardToken = await deployERC20Token();

    const morFactory = await ethers.getContractFactory('MOR');
    mor = await morFactory.deploy(wei(1000000)); // 1M initial supply

    // Deploy swap router mocks
    const swapRouterFactory = await
ethers.getContractFactory('UniswapSwapRouterMock');
    swapRouter = await swapRouterFactory.deploy();

    const nftManagerFactory = await
ethers.getContractFactory('NonfungiblePositionManagerMock');
    nonfungiblePositionManager = await nftManagerFactory.deploy();

    // Deploy bridge mocks
    const bridgeFactory = await
ethers.getContractFactory('ArbitrumBridgeGatewayRouterMock');
    arbitrumBridge = await bridgeFactory.deploy();

    // Deploy LayerZero mocks
    const lzFactory = await
ethers.getContractFactory('LayerZeroEndpointV2Mock');
    lzEndpointL1 = await lzFactory.deploy(l1ChainId, owner.address);
    lzEndpointL2 = await lzFactory.deploy(l2ChainId, owner.address);

    // Deploy L2MessageReceiver mock (if needed for L1Sender tests)
    const l2MessageReceiverFactory = await
ethers.getContractFactory('L2MessageReceiver');
    const l2MessageReceiverImpl = await
l2MessageReceiverFactory.deploy();
    const proxyFactory = await ethers.getContractFactory('ERC1967Proxy');
    const l2MessageReceiverProxy = await
proxyFactory.deploy(l2MessageReceiverImpl, '0x');
    l2MessageReceiver =
l2MessageReceiverFactory.attach(l2MessageReceiverProxy) as
L2MessageReceiver;
    await l2MessageReceiver.L2MessageReceiver__init();
  }

  async function deployCapitalProtocol() {
    // 1. Deploy ChainLinkDataConsumer
    const chainLinkFactory = await
ethers.getContractFactory('ChainLinkDataConsumer');
    const chainLinkImpl = await chainLinkFactory.deploy();
    const proxyFactory = await ethers.getContractFactory('ERC1967Proxy');
    const chainLinkProxy = await proxyFactory.deploy(await
chainLinkImpl.getAddress(), '0x');
    chainLinkDataConsumer = chainLinkFactory.attach(await
chainLinkProxy.getAddress()) as ChainLinkDataConsumer;
    await chainLinkDataConsumer.ChainLinkDataConsumer_init();

    // Setup price feeds
    await chainLinkDataConsumer.updateDataFeeds(
      ['ETH/USD', 'wBTC/BTC,BTC/USD'],
      [[await ethUsdFeed.getAddress()], [await wbtcBtcFeed.getAddress(),
await btcUsdFeed.getAddress()]],
    );

    // 2. Deploy RewardPool (with library linking)
    const linearDistributionLib = await (
      await
ethers.getContractFactory('LinearDistributionIntervalDecrease')
    ).deploy();
    const rewardPoolFactory = await
ethers.getContractFactory('RewardPool', {
      libraries: {
        LinearDistributionIntervalDecrease: await
linearDistributionLib.getAddress(),
      },
    });
    const rewardPoolImpl = await rewardPoolFactory.deploy();
    const rewardPoolProxy = await proxyFactory.deploy(await
rewardPoolImpl.getAddress(), '0x');
    rewardPool = rewardPoolFactory.attach(await
rewardPoolProxy.getAddress()) as RewardPool;

    const pools: IRewardPool.RewardPoolStruct[] = [
      {
        payoutStart: oneDay,
        decreaseInterval: oneDay,
        initialReward: 10000000,
        rewardDecrease: 1000000,
        isPublic: true,
      },
      {
        payoutStart: oneDay * 20,
        decreaseInterval: oneDay * 2,
        initialReward: wei(200),
        rewardDecrease: wei(1),
        isPublic: false,
      },
    ];
    await rewardPool.RewardPool_init(pools);

    // 3. Deploy L1SenderV2
    const l1SenderFactory = await
ethers.getContractFactory('L1SenderV2');
    const l1SenderImpl = await l1SenderFactory.deploy();
    const l1SenderProxy = await proxyFactory.deploy(await
l1SenderImpl.getAddress(), '0x');
    l1Sender = l1SenderFactory.attach(await l1SenderProxy.getAddress())
as L1SenderV2;
    await l1Sender.L1SenderV2__init();

    // Configure L1Sender
    await l1Sender.setStETh(await stETH.getAddress());
    await l1Sender.setArbitrumBridgeConfig({
      wstETH: await wstETH.getAddress(),
      gateway: await arbitrumBridge.getAddress(),
      receiver: treasury.address,
    });
    await l1Sender.setUniswapSwapRouter(await swapRouter.getAddress());

    // 4. Deploy Distributor
    const distributorFactory = await
ethers.getContractFactory('Distributor');
    const distributorImpl = await distributorFactory.deploy();
    const distributorProxy = await proxyFactory.deploy(await
distributorImpl.getAddress(), '0x');
    distributor = distributorFactory.attach(await
distributorProxy.getAddress()) as Distributor;
    await distributor.Distributor_init(
      await chainLinkDataConsumer.getAddress(),
      await aavePool.getAddress(),
      await aavePoolDataProvider.getAddress(),
      await rewardPool.getAddress(),
      await l1Sender.getAddress(),
    );

    // 5. Deploy DepositPool
    const lib1 = await (await
ethers.getContractFactory('ReferrerLib')).deploy();
```

```
    const lib2 = await (await
ethers.getContractFactory('LockMultiplierMath')).deploy();
    const depositPoolFactory = await
ethers.getContractFactory('DepositPool', {
      libraries: {
        ReferrerLib: await lib1.getAddress(),
        LockMultiplierMath: await lib2.getAddress(),
      },
    });
    const depositPoolImpl = await depositPoolFactory.deploy();
    const depositPoolProxy = await proxyFactory.deploy(await
depositPoolImpl.getAddress(), '0x');
    depositPool = depositPoolFactory.attach(await
depositPoolProxy.getAddress()) as DepositPool;
    await depositPool.DepositPool_init(await depositToken.getAddress(),
await distributor.getAddress());

    // 6. Deploy L2TokenReceiverV2
    const l2ReceiverFactory = await
ethers.getContractFactory('L2TokenReceiverV2');
    const l2ReceiverImpl = await l2ReceiverFactory.deploy();
    const l2ReceiverProxy = await proxyFactory.deploy(await
l2ReceiverImpl.getAddress(), '0x');
    l2TokenReceiver = l2ReceiverFactory.attach(await
l2ReceiverProxy.getAddress()) as L2TokenReceiverV2;
    await l2TokenReceiver.L2TokenReceiver__init(
      await swapRouter.getAddress(),
      await nonfungiblePositionManager.getAddress(),
      {
        tokenIn: await stETH.getAddress(),
        tokenOut: await mor.getAddress(),
        fee: 500,
        sqrtPriceLimitX96: 0,
      },
    );

    // 7. Setup connections
    await l1Sender.setDistributor(await distributor.getAddress());

    // Add deposit pool to distributor
    await distributor.addDepositPool(
      publicRewardPoolId,
      await depositPool.getAddress(),
      await depositToken.getAddress(),
      'ETH/USD',
      Strategy.NONE,
    );

    // Complete migration
    await depositPool.migrate(publicRewardPoolId);
  }

  async function mintTokensToUsers() {
    const amount = wei(1000);

    // Mint deposit tokens
    await depositToken.mint(alice.address, amount);
    await depositToken.mint(bob.address, amount);
    await depositToken.mint(charlie.address, amount);

    // Mint stETH
    await stETH.mint(alice.address, amount);
    await stETH.mint(bob.address, amount);
    await stETH.mint(charlie.address, amount);

    // Setup approvals
    await depositToken.connect(alice).approve(await
depositPool.getAddress(), ethers.MaxUint256);
    await depositToken.connect(bob).approve(await
depositPool.getAddress(), ethers.MaxUint256);
    await depositToken.connect(charlie).approve(await
depositPool.getAddress(), ethers.MaxUint256);

    await stETH.connect(alice).approve(await l1Sender.getAddress(),
ethers.MaxUint256);
    await stETH.connect(bob).approve(await l1Sender.getAddress(),
ethers.MaxUint256);
  }

  // Test Setup
  before(async function () {
    // Get signers
    [owner, alice, bob, charlie, treasury, referrer1, referrer2] = await
ethers.getSigners();

    // Deploy all contracts
    await deployMocks();
    await deployCapitalProtocol();
    await mintTokensToUsers();

    // Take snapshot for reverting
    await reverter.snapshot();
  });

  afterEach(async function () {
    await reverter.revert();
  });

  describe('POC yield not withdrawable', function () {
    it('should show yield stuck', async function () {
      await setTime(oneDay * 9);

      // Setup reward pool timestamp (required before any staking)
      await
distributor.setRewardPoolLastCalculatedTimestamp(publicRewardPoolId,
oneDay * 7);

      // Set minimum rewards distribute period to 1 day
      await distributor.setMinRewardsDistributePeriod(1);

      const timestamp = await
distributor.rewardPoolLastCalculatedTimestamp(publicRewardPoolId);
      expect(timestamp).to.eq(604800);

      const dp0Before = await
distributor.depositPools(publicRewardPoolId, depositPool);
      expect(dp0Before.deposited).to.eq(wei(0));
      expect(dp0Before.lastUnderlyingBalance).to.eq(wei(0));

      // Alice stakes tokens
      await depositPool.connect(alice).stake(publicRewardPoolId,
wei(100), 0, ZERO_ADDR);

      const dp0 = await distributor.depositPools(publicRewardPoolId,
depositPool);
      expect(dp0.deposited).to.eq(wei(100));
      expect(dp0.lastUnderlyingBalance).to.eq(wei(100));

      const currentUserRewards = await
depositPool.connect(alice).getLatestUserReward(publicRewardPoolId,
alice);
      expect(currentUserRewards).to.eq(0);

      // imitate yield
      await depositToken.mint(distributor, wei(100));

      expect(await depositToken.balanceOf(distributor)).to.eq(wei(200));
      const rewards = await rewardPool.getPeriodRewards(0, oneDay * 9,
oneDay * 11);
      expect(rewards).to.eq(3000000);

      // Fast forward time to maxEndTime
      await setTime(oneDay * 11);

      // rewards are distributed at that time and yield withdrawn
      await distributor.distributeRewards(publicRewardPoolId);

      // lastUnderlyingBalance updated
      const dp1 = await distributor.depositPools(publicRewardPoolId,
depositPool);
      expect(dp1.deposited).to.eq(wei(100));
      expect(dp1.lastUnderlyingBalance).to.eq(wei(200));

      // yield is withdrawn
      await distributor.withdrawYield(publicRewardPoolId, depositPool);
      expect(await depositToken.balanceOf(distributor)).to.eq(wei(100));
      expect(await depositToken.balanceOf(l1Sender)).to.eq(wei(100));

      // lastUnderlyingBalance updated
      const dp2 = await distributor.depositPools(publicRewardPoolId,
depositPool);
      expect(dp2.deposited).to.eq(wei(100));
      expect(dp2.lastUnderlyingBalance).to.eq(wei(100));

      // alice receives rewards
      const currentUserRewardsFinal = await
depositPool.connect(alice).getLatestUserReward(publicRewardPoolId,
alice);
      expect(currentUserRewardsFinal).to.be.gt(0);

      // Alice withdraws at oneDay * 14
      await setTime(oneDay * 14);

      // imitate yield that accrues after all reward distributed, they
will never be tracked
      await depositToken.mint(distributor, wei(100));

      // rewards for this period are 0
      const rewardsFinal = await rewardPool.getPeriodRewards(0, oneDay *
11, oneDay * 14);
      expect(rewardsFinal).to.eq(0);

      // lastUnderlyingBalance still the same
      const dp3 = await distributor.depositPools(publicRewardPoolId,
depositPool);
      expect(dp3.deposited).to.eq(wei(100));
      expect(dp3.lastUnderlyingBalance).to.eq(wei(100));

      await depositPool.connect(alice).withdraw(publicRewardPoolId,
wei(100));
      expect(await depositToken.balanceOf(distributor)).to.eq(wei(100));

      // lastUnderlyingBalance not updated, does not track yield
      const dp4 = await distributor.depositPools(publicRewardPoolId,
depositPool);
      expect(dp4.deposited).to.eq(wei(0));
      expect(dp4.lastUnderlyingBalance).to.eq(wei(0));

      // yield cannot be withdrawn
      await distributor.withdrawYield(publicRewardPoolId, depositPool);
      expect(await depositToken.balanceOf(distributor)).to.eq(wei(100));

      // l1Sender balance still the same -> no yield withdrawn
      expect(await depositToken.balanceOf(l1Sender)).to.eq(wei(100));
    });
  });
});
```

Low Risk and Non-Critical Issues

For this audit, 26 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xrex** received the top score from the judge.

The following wardens also submitted reports: 0xAura, 0xcb90f054, 0xdoichantran, Ahmerdrarerh, arjun16, befree3x, blaze18, cerweb10, hypna, K42, KlosMitSoss, ks__xxxxx, LinKenji, Maketer7, milnail, n0m4d1c_b34r, newspacexyz, osok, PolarizedLight, rayss, Sancybars, SOPROBRO, Sparrow, Teycir, and Yaneca_b.

[01] An attacker can force the distributor to mint MOR rewards in a period of no new generated yield

The `distributeRewards()` function of the Distributor contract determines reward amounts to be distributed to DepositPool appropriately. The reward token minted is MOR tokens in this case. In the current implementation of the onchain distribution contracts, over 3k MOR is being minted/distributed daily.

The issue is that once we upgrade the contracts to v7, each deposit pool will now have a portion of MOR based on the yield they contributed from timestamp x to timestamp y. This leaves room for an attacker to force the Distributor into minting MOR tokens even on days where there was no actual yield gains.

https://github.com/code-423n4/2025-08-morpheus/blob/main/contracts/capital-protocol/Distributor.sol#L330-L410

```
 function distributeRewards(uint256 rewardPoolIndex_) public {
        //// Base validation
        IRewardPool rewardPool_ = IRewardPool(rewardPool);
        rewardPool_.onlyExistedRewardPool(rewardPoolIndex_);

        uint128 lastCalculatedTimestamp_ =
rewardPoolLastCalculatedTimestamp[rewardPoolIndex_];
        require(lastCalculatedTimestamp_ != 0, "DR:
`rewardPoolLastCalculatedTimestamp` isn't set");
        //// End

        //// Calculate the reward amount
        uint256 rewards_ = IRewardPool(rewardPool).getPeriodRewards(
            rewardPoolIndex_,
            lastCalculatedTimestamp_,
            uint128(block.timestamp)
        );

        if (rewards_ == 0) return;
        //// End

        // Stop execution when the reward pool is private
        if (!rewardPool_.isRewardPoolPublic(rewardPoolIndex_)) {
            _onlyExistedDepositPool(rewardPoolIndex_,
depositPoolAddresses[rewardPoolIndex_][0]);
            distributedRewards[rewardPoolIndex_]
[depositPoolAddresses[rewardPoolIndex_][0]] += rewards_;

            rewardPoolLastCalculatedTimestamp[rewardPoolIndex_] =
uint128(block.timestamp);

            return;
        }

        // Validate that public reward pools await
`minRewardsDistributePeriod`
        if (block.timestamp <= lastCalculatedTimestamp_ +
minRewardsDistributePeriod) return;
```

```
        rewardPoolLastCalculatedTimestamp[rewardPoolIndex_] =
uint128(block.timestamp);

        //// Update prices
        updateDepositTokensPrices(rewardPoolIndex_);
        //// End

        //// Calculate `yield` from all deposit pools
        uint256 length_ = depositPoolAddresses[rewardPoolIndex_].length;
        uint256 totalYield_ = 0;
        uint256[] memory yields_ = new uint256[](length_);

        for (uint256 i = 0; i < length_; i++) {
            DepositPool storage depositPool =
depositPools[rewardPoolIndex_][depositPoolAddresses[rewardPoolIndex_]
[i]];

            address yieldToken_;
            if (depositPool.strategy == Strategy.AAVE) {
                yieldToken_ = depositPool.aToken;
            } else if (depositPool.strategy == Strategy.NONE) {
                // The current condition coverage cannot be achieved in
the current version.
                // Added to avoid errors in the future.
                yieldToken_ = depositPool.token;
            }

@>            uint256 balance_ =
IERC20(yieldToken_).balanceOf(address(this));
            uint256 decimals_ = IERC20Metadata(yieldToken_).decimals();
@>            uint256 underlyingYield_ = (balance_ -
depositPool.lastUnderlyingBalance).to18(decimals_);
            uint256 yield_ = underlyingYield_ * depositPool.tokenPrice;

            depositPool.lastUnderlyingBalance = balance_;

            yields_[i] = yield_;
            totalYield_ += yield_;
        }

        if (totalYield_ == 0) {
            undistributedRewards += rewards_;
            return;
        }
        //// End
        //// Calculate `depositPools` shares and reward amount for each
`depositPool`
        for (uint256 i = 0; i < length_; i++) {
            if (yields_[i] == 0) continue;

            distributedRewards[rewardPoolIndex_]
[depositPoolAddresses[rewardPoolIndex_][i]] +=
                (yields_[i] * rewards_) /
                totalYield_;
        }
        //// End
    }
```

Proper scenario:

1. DepositPool A is connected to reward pool 0 which is a public reward pool and have Aave strategy connected to it in the Distributor when we call `addDepositPool()`.
2. Users deposit 10k wstETH or wBTC etc on day 1. 10k wBTC is minted to the distributor contract.
3. On day 2, the total shares grow from 10k to 10.01 wBTC because now the liquidity index in Aave have grown, interest has accrued etc but the idea is that from day 1 to day 2, the distributor has earned 0.01 wBTC yield.
4. Now, the distributor will mint e.g 3k MOR tokens for that day in which yield was earned and users each get a claim of that based on their stake.

Attack scenario:

1. Suppose the strategy in this case is not Aave but rather another strategy such as `NONE`. This means there is a strategy, it is not `NO_YIELD` but it also isn't Aave. For example, stETH would be one such strategy.
2. If users stake 10k stETH on day 1, then a slash occur that forces stETH to rebase negative slightly, let's assume a -0.001 stETH.
3. On day 2, the shares/balance of the distributor in stETH becomes 10k stETH - -0.001 stETH, this means no yield and instead a negative rebase and users would have to wait for another day or 2 for the rebase to become positive again and exceed total deposit (10k stETH) before MOR tokens being minted will then resume
4. However, an attacker can donate 0.0011 stETH to the distributor and force all 3k MOR tokens to be minted for that zero-yield day.

[02] For pools which have stETH as the deposit token, switching to Aave strategy will not work

In the current implementation of the Distribution contracts onchain, only stETH have been staked by users thereby earning MOR in return. However, if the protocol were to try to switch to Aave strategy for these staked tokens, it would fail.

https://github.com/code-423n4/2025-08-morpheus/blob/main/contracts/capital-protocol/Distributor.sol#L192-L248

```
 function addDepositPool(
        uint256 rewardPoolIndex_,
        address depositPoolAddress_,
        address token_,
        string memory chainLinkPath_,
        Strategy strategy_
    ) external onlyOwner {
        IRewardPool rewardPool_ = IRewardPool(rewardPool);
        rewardPool_.onlyExistedRewardPool(rewardPoolIndex_);

        require(

IERC165(depositPoolAddress_).supportsInterface(type(IDepositPool).interfa
ceId),
            "DR: the deposit pool address is invalid"
        );

        // Validate that pool is public in other cases.
        if (strategy_ == Strategy.NO_YIELD) {
            // Validate that pool is private.
            rewardPool_.onlyNotPublicRewardPool(rewardPoolIndex_);
            // Validate that deposit pool is not added for this
`rewardPoolIndex_`.
            require(
                depositPoolAddresses[rewardPoolIndex_].length == 0,
                "DR: the deposit pool for this index already added"
            );

            // Skip `token_` and `chainLinkPath_` when
`Strategy.NO_YIELD`.
            token_ = address(0);
            chainLinkPath_ = "";
        } else {
            require(!isDepositTokenAdded[token_], "DR: the deposit token
already added");

            rewardPool_.onlyPublicRewardPool(rewardPoolIndex_);
        }

        // Set `aToken_` when `Strategy.AAVE`. Add allowance for Aave to
transfer `token_` from the current
```

```
        // contract.
        address aToken_ = address(0);
        if (strategy_ == Strategy.AAVE) {
            (aToken_, , ) =
AaveIPoolDataProvider(aavePoolDataProvider).getReserveTokensAddresses(tok
en_);

            IERC20(token_).safeApprove(aavePool, type(uint256).max);
            IERC20(aToken_).approve(aavePool, type(uint256).max);
        }

        DepositPool memory depositPool_ = DepositPool(token_,
chainLinkPath_, 0, 0, 0, strategy_, aToken_, true);

        depositPoolAddresses[rewardPoolIndex_].push(depositPoolAddress_);
        depositPools[rewardPoolIndex_][depositPoolAddress_] =
depositPool_;
        isDepositTokenAdded[token_] = true;

        // Update prices for all `depositPools` by `rewardPoolIndex_`
        if (strategy_ != Strategy.NO_YIELD) {
            updateDepositTokensPrices(rewardPoolIndex_);
        }

        emit DepositPoolAdded(rewardPoolIndex_, depositPool_);
    }
```

One newer features of the v7 contracts such as DepositPool.sol is that it now sends the staked tokens to the Distributor contract. This allows the protocol to also have other yield strategies such as supplying user staked tokens to Aave markets.

1. If we were to upgrade from v6 to v7 for DepositPool A for the reward pool index of 0 which is a public pool.
2. Then, call `addRewardPool` in the RewardPool contract to add the reward pool index of 0.
3. Next call, `addDepositPool()` in the Distributor contract to whitelist reward pool index 0, it would fail.
4. The reason is because in the currently deployed v6 Distribution contracts (which the protocol will upgrade to use DepositPool implementation), only stETH is the token staked. Thus, the Distributor contract trying to call approve on the `aToken` address will fail. Because on Aave, there is no corresponding `aToken` for the stETH deposit token and for that reason, Aave will return the zero address which the contract will then try to call approve on.

[03] Any minimum reward distribution period for stETH pools that is less than or equal to 24 hrs can be forced to be non-distributed by an attacker

In the current deployments of the v6 contracts (Distribution) onchain, the minimum stake amount is 0.01 stETH. Combined with the daily timeframe Lido reports and distribute's rewards, an attacker could force the Distributor to not mint MOR token rewards for users from timestamp x to timestamp z.

```
 function distributeRewards(uint256 rewardPoolIndex_) public {
        ...
        //// Calculate the reward amount
@>        uint256 rewards_ = IRewardPool(rewardPool).getPeriodRewards(
            rewardPoolIndex_,
            lastCalculatedTimestamp_,
            uint128(block.timestamp)
        );

        if (rewards_ == 0) return;
        //// End

       ...

        // Validate that public reward pools await
`minRewardsDistributePeriod`
@>        if (block.timestamp <= lastCalculatedTimestamp_ +
minRewardsDistributePeriod) return;
        rewardPoolLastCalculatedTimestamp[rewardPoolIndex_] =
uint128(block.timestamp);

        //// Update prices
        updateDepositTokensPrices(rewardPoolIndex_);
        //// End

        //// Calculate `yield` from all deposit pools
        uint256 length_ = depositPoolAddresses[rewardPoolIndex_].length;
        uint256 totalYield_ = 0;
        uint256[] memory yields_ = new uint256[](length_);

        for (uint256 i = 0; i < length_; i++) {
            DepositPool storage depositPool =
depositPools[rewardPoolIndex_][depositPoolAddresses[rewardPoolIndex_]
[i]];

            address yieldToken_;
            if (depositPool.strategy == Strategy.AAVE) {
                yieldToken_ = depositPool.aToken;
            } else if (depositPool.strategy == Strategy.NONE) {
                // The current condition coverage cannot be achieved in
the current version.
                // Added to avoid errors in the future.
                yieldToken_ = depositPool.token;
            }

            uint256 balance_ =
IERC20(yieldToken_).balanceOf(address(this));
            uint256 decimals_ = IERC20Metadata(yieldToken_).decimals();
            uint256 underlyingYield_ = (balance_ -
depositPool.lastUnderlyingBalance).to18(decimals_);
            uint256 yield_ = underlyingYield_ * depositPool.tokenPrice;

            depositPool.lastUnderlyingBalance = balance_;

            yields_[i] = yield_;
            totalYield_ += yield_;
        }

        if (totalYield_ == 0) {
 @>           undistributedRewards += rewards_;
            return;
        }

        ...
    }
```

1. Lido distributes rewards each day at 00:00 UTC (aka every 24 hours)
2. DepositPool A has 10k stETH staked by users
3. The `minRewardsDistributePeriod` is set to 1 hour for example
4. What an attacker can do is to stake just a little over the minimum stake (0.0105) since minimum is 0.01 stETH
5. Say 1 hours has elapsed, call the `distributeRewards()` directly or trigger a transaction such as `withdraw()` from the DepositPool to withdraw 1 wei stETH which will then trigger `distributeRewards()`
6. What happens is that if the reward pool index e.g 0 mints 3500 MOR rewards per day, that means we do 3500 / 24. Thus, `uint256 rewards_ = IRewardPool(rewardPool).getPeriodRewards(rewardPoolIndex_, lastCalculatedTimestamp_, uint128(block.timestamp));` will return 145.83 MOR, but since it is not yet past 00:00 UTC and stETH shares / balances have not gone up, these 145.83 MOR will not be minted and instead go into `undistributedRewards`.

[04] If a deposit pool has more than 1 public pools connected to it, issues will occur for some users after migration

Deposit pools can have more than 1 reward pools attached to it. In the case whereby we have 2 reward `rewardPoolIndex_` for Deposit pool A, and there is some staked tokens in each of these reward pool indexes, after migration, some users will be locked out of withdrawing.

https://github.com/code-423n4/2025-08-morpheus/blob/main/contracts/capital-protocol/DepositPool.sol#L137-L160

```
function migrate(uint256 rewardPoolIndex_) external onlyOwner {
        require(!isMigrationOver, "DS: the migration is over");
        if (totalDepositedInPublicPools == 0) {
            isMigrationOver = true;
            emit Migrated(rewardPoolIndex_);

            return;
        }

        IRewardPool rewardPool_ =
IRewardPool(IDistributor(distributor).rewardPool());
        rewardPool_.onlyExistedRewardPool(rewardPoolIndex_);
        rewardPool_.onlyPublicRewardPool(rewardPoolIndex_);

        // Transfer yield to prevent the reward loss
        uint256 remainder_ =
IERC20(depositToken).balanceOf(address(this)) -
totalDepositedInPublicPools;
        require(remainder_ > 0, "DS: yield for token is zero");

        IERC20(depositToken).transfer(distributor, remainder_);

        IDistributor(distributor).supply(rewardPoolIndex_,
totalDepositedInPublicPools);

        isMigrationOver = true;

        emit Migrated(rewardPoolIndex_);
    }
```

Suppose there are 2 public reward pool indexes of 0 & 1. And we have DepositPool A where users have staked stETH.

1. stETH staked on reward pool index 0 is 500

2. stETH staked on reward pool index 1 is also 500. The `totalDepositedInPublicPools` in this case would be 1000 stETH
3. When we call the `migrate()` function, it would send 1000 stETH to the Distributor contract on behalf of whatever `rewardPoolIndex_` we call the `migrate()` function with. That is to say, if we call the function with `rewardPoolIndex_` args of 0, then it will deposit all 1k stETH for users of `rewardPoolIndex_` 0. But the total staked by those users is 500 stETH where the users of `rewardPoolIndex_` 1 contributed the rest 500 stETH.
4. In this case, 500 stETH will be lost by users of `rewardPoolIndex_` 1 as these cannot be withdrawn since we supplied all 1k stETH on `rewardPoolIndex_` 0 in the Distributor contract.

It is better to migrate the amount deposited by users of `rewardPoolIndex_` x when the `migrate` function is called. This way, the migrations of 0 & 1 will accurately supply 500 stETH each in the Distributor contract for users.

[05] Non-utilized `isPrivateDepositPoolAdded` variable in the Distributor contract

In the Distributor contract, there is a `isPrivateDepositPoolAdded` variable that is supposed to track if a reward pool index attached to private deposit pools is known in the contract i.e, it's the same emulator for the `_onlyExistedDepositPool` function but specifically for private deposit pools in this case.

https://github.com/code-423n4/2025-08-morpheus/blob/main/contracts/capital-protocol/Distributor.sol#L39

```
mapping(uint256 => uint128) public rewardPoolLastCalculatedTimestamp;
    // @audit QA un-used variable
@>    mapping(uint256 => bool) public isPrivateDepositPoolAdded;
```

Since, this variable is not being used in the Distributor contract at this time, it should be removed.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.