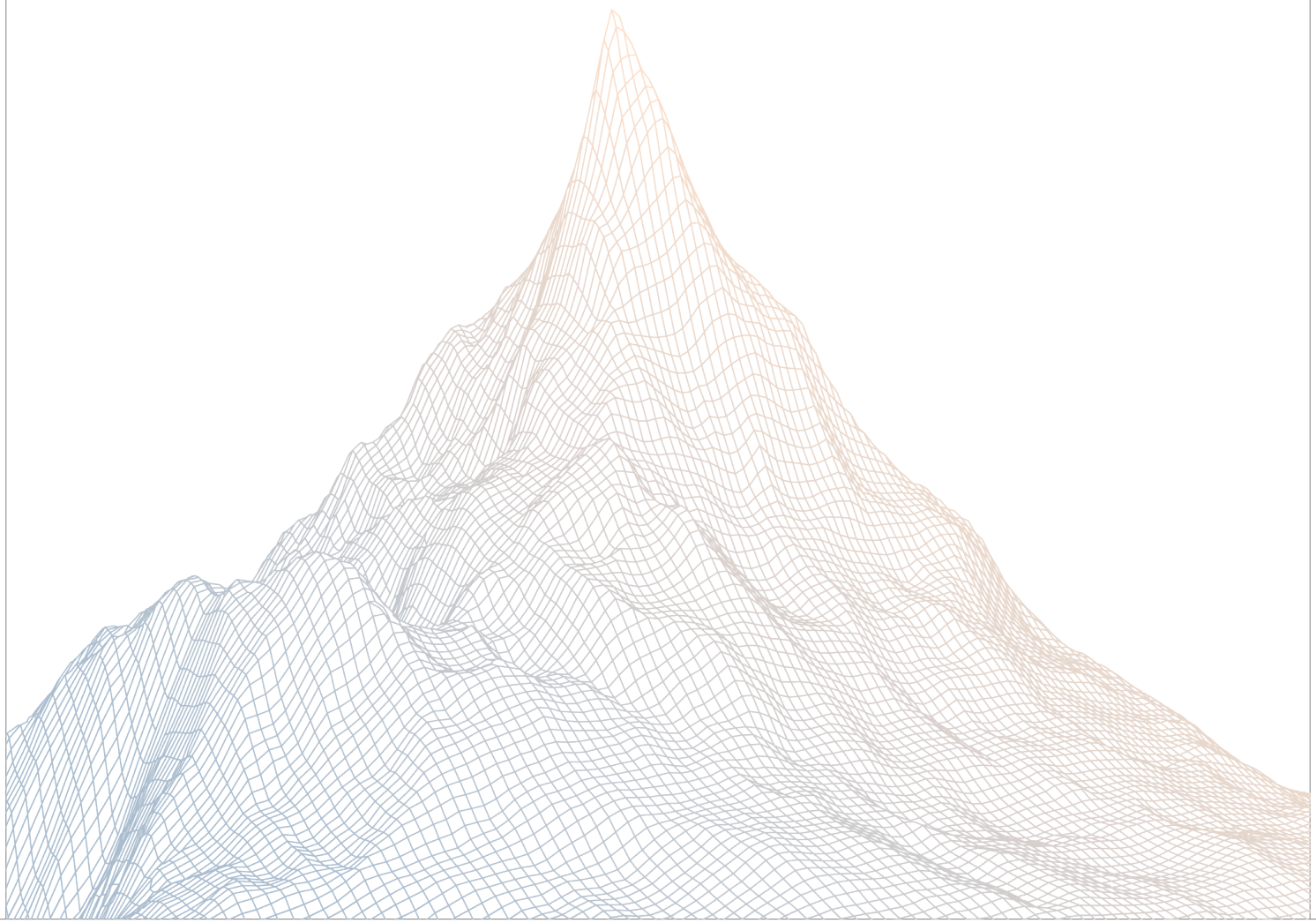
Zenith

# Morpheus

## Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: July 22th to July 25th, 2025

AUDITED BY: rscodes
said

# Contents

# 1 Introduction

## 1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

## 1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

## 1.3 Risk Classification

| SEVERITY LEVEL | IMPACT: HIGH | IMPACT: MEDIUM | IMPACT: LOW |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

# 2

# Executive Summary

## 2.1 About Morpheus

Morpheus is a Decentralized AI network designed to incentivize a network of Smart Agents, general-purpose AI that can execute Smart Contracts on behalf of users. It utilizes LLM (Large Language Model) Transformer technology to convert human language into actionable code for Web3 interactions, using intent-based abstractions.

The objective of Morpheus is to allow anyone who wants to deploy a Smart Agent or Decentralized AI application to have the compute, capital, code and community rails they need to do so.

## 2.2 Scope

The engagement involved a review of the following targets:

| | |
|---|---|
| **Target** | SmartContracts |
| **Repository** | https://github.com/MorpheusAIs/SmartContracts |
| **Commit Hash** | `091b6f7ecbb2933c9d1a805bd4e4c926a02f38a6` |
| **Files** | `contracts/capital-protocol/* (except DistributionV6.sol)` |

## 2.3 Audit Timeline

| | |
|---|---|
| **July 22, 2025** | Audit start |
| **July 25, 2025** | Audit end |
| **July 31, 2025** | Draft report published |

## 2.4 Issues Found

| SEVERITY | COUNT |
|---|---|
| Critical Risk | 1 |
| High Risk | 1 |
| Medium Risk | 2 |
| Low Risk | 4 |
| Informational | 2 |
| **Total Issues** | **10** |

# 3

## Findings Summary

| ID | Description | Status |
|---|---|---|
| C-1 | withdraw functionality will not work with the Aave strategy | Resolved |
| H-1 | Updating rewardPoolLastCalculatedTimestamp early allows a griefer to deny reward distribution | Resolved |
| M-1 | Attackers can potentially halt staking in some deposit pools | Acknowledged |
| M-2 | Using the same token in multiple DepositPools causes incorrect yield accounting | Resolved |
| L-1 | Chainlink may return stale prices leading to incorrect token valuation | Resolved |
| L-2 | Updating rewardPoolLastCalculatedTimestamp with zero rewards enables griefing attack | Resolved |
| L-3 | Migration will fail if the pool has no yield | Acknowledged |
| L-4 | Self-referring is possible and can potentially be abused by users | Acknowledged |
| I-1 | Calculation method can be improved to eliminate any chance of overflow | Resolved |
| I-2 | Missing claim for Aave incentives inside Distributor leads to unclaimed rewards | Resolved |

# 4

Findings

## 4.1 Critical Risk

A total of 1 critical risk findings were identified.

### [C-1] `withdraw` functionality will not work with the Aave strategy

| SEVERITY: Critical | IMPACT: High |
| --- | --- |
| STATUS: Resolved | LIKELIHOOD: High |

#### Target

- DepositPool.sol#L455

#### Description:

When `withdraw` is requested, it checks the requested `amount_` against the `depositToken` balance in the Distributor. If it is greater than the balance, the amount is set to the balance.

- DepositPool.sol#L455

```
    function _withdraw(address user_, uint256 rewardPoolIndex_,
    uint256 amount_, uint256 currentPoolRate_) private {
        require(isMigrationOver ==
         true, "DS: migration isn't over");

        RewardPoolProtocolDetails storage rewardPoolProtocolDetails
    = rewardPoolsProtocolDetails[rewardPoolIndex_];
        RewardPoolData storage rewardPoolData
    = rewardPoolsData[rewardPoolIndex_];
        UserData storage userData = usersData[user_][rewardPoolIndex_];

        uint256 deposited_ = userData.deposited;
        require(deposited_ > 0, "DS: user isn't staked");

        if (amount_ > deposited_) {
            amount_ = deposited_;
        }

        uint256 newDeposited_;
     if (IRewardPool
```

```
      (IDistributor(distributor)
        .rewardPool())
          .isRewardPoolPublic
            (rewardPoolIndex_)) {
            require(
                block.timestamp > userData.lastStake
    + rewardPoolProtocolDetails.withdrawLockPeriodAfterStake,
                "DS: pool withdraw is locked"
            );

>>>         uint256 depositTokenContractBalance_
    = IERC20(depositToken).balanceOf(distributor);
            if (amount_ > depositTokenContractBalance_) {
                amount_ = depositTokenContractBalance_;
            }

            // ...
    }
```

However, in the `Distributor` contract, if the `RewardPool` strategy is `AAVE`, it supplies all `depositToken` to Aave.

- Distributor.sol#L273-L275

```
function supply
(uint256 rewardPoolIndex_, uint256 amount_) external {
    address depositPoolAddress_ = _msgSender();
    _onlyExistedDepositPool(rewardPoolIndex_, depositPoolAddress_);

    DepositPool storage depositPool
    = depositPools[rewardPoolIndex_][depositPoolAddress_];
    require(depositPool.strategy != Strategy.NO_YIELD, "DR: invalid strategy
    for the deposit pool");

    distributeRewards(rewardPoolIndex_);
    _withdrawYield(rewardPoolIndex_, depositPoolAddress_);

    IERC20(depositPool.token).safeTransferFrom(depositPoolAddress_,
    address(this), amount_);
    if (depositPool.strategy == Strategy.AAVE) {
AaveIPool(aavePool).supply(depositPool.token, amount_, address(this), 0);
    }

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

And there is no functionality in the `Distributor` to pull funds from Aave, other than `withdraw`, which causes user funds to be stuck and unwithdrawable, since the `depositToken` balance in the `Distributor` will be zero.

### Recommendations:

Instead of using the `depositToken` balance, `depositPool.deposited` or the `aToken` balance of the Distributor can be used.

**Morpheus:** Resolved with @ff56ece2d60a...

**Zenith:** Verified.

## 4.2 High Risk

A total of 1 high risk findings were identified.

### [H-1] Updating `rewardPoolLastCalculatedTimestamp` early allows a griefer to deny reward distribution

| | |
|---|---|
| SEVERITY: High | IMPACT: High |
| STATUS: Resolved | LIKELIHOOD: Medium |

#### Target

- Distributor.sol#L322

#### Description:

Inside `distributeRewards`, `rewardPoolLastCalculatedTimestamp` is updated to `block.timestamp` before the `minRewardsDistributePeriod` check is performed.

- Distributor.sol#L322

```
    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)
        );
>>>     rewardPoolLastCalculatedTimestamp[rewardPoolIndex_]
    = 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_;

            return;
        }

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

        // ...
    }
```

If `block.timestamp` is less than or equal to `lastCalculatedTimestamp_ + minRewardsDistributePeriod`, the operation will return early without distributing rewards, even though `rewardPoolLastCalculatedTimestamp` has already been updated.

This opens a griefing attack vector, where an attacker repeatedly calls `distributeRewards` before `minRewardsDistributePeriod` has passed, updating `rewardPoolLastCalculatedTimestamp` and preventing proper reward distribution.

## Recommendations:

Update `rewardPoolLastCalculatedTimestamp` only after passing the `minRewardsDistributePeriod` check.

**Morpheus:** Resolved with @406ad354f30...

**Zenith:** Verified.

## 4.3 Medium Risk

A total of 2 medium risk findings were identified.

### [M-1] Attackers can potentially halt staking in some deposit pools

| | |
|---|---|
| SEVERITY: Medium | IMPACT: Medium |
| STATUS: Acknowledged | LIKELIHOOD: Low |

#### Target

- DepositPool.sol

#### Description:

In `stake` of `DepositPool.sol`, the code uses `rewardPoolsProtocolDetails[rewardPoolIndex_].distributedRewards` to account for the total amount of tokens that were distributed as rewards so far.

In `_getCurrentPoolRate` we can see that the `rewards_` that get returned to be added to the `.distributedRewards` is:

```
uint256 rewards_
    = IDistributor(distributor).getDistributedRewards(rewardPoolIndex_,
    address(this)) -
            rewardPoolsProtocolDetails[rewardPoolIndex_].distributedRewards;
```

Next we notice that there is no reentrancy guard in `stake` and in `_stake` there is an external call:

```
IERC20(depositToken).safeTransferFrom(_msgSender(), address(this), amount_);
```

Hence if the token has the callback, attackers can exploit the following sequence:

1. currently accounted for rewards = 10
2. Attacker calls stake
3. New distributed becomes 15 hence rewards is returned as 15 - 10 = 5.

4. But in `_stake` attacker gets an reentrant call.
5. Attacker calls stake again (in the callback)
6. Rewards returned calculated in `_getCurrentPoolRate` will still be 5
7. `rewardPoolsProtocolDetails[rewardPoolIndex_].distributedRewards` += 5
8. Callback ends, original trace continues
9. `rewardPoolsProtocolDetails[rewardPoolIndex_].distributedRewards` += 5 (again)

Now `rewardPoolsProtocolDetails[rewardPoolIndex_].distributedRewards` **is more than** `IDistributor(distributor).getDistributedRewards(rewardPoolIndex_, address(this))`.

This means the internal function `_getCurrentPoolRate` called by `stake` will revert due to it being negative **from the next call onwards**:

```
uint256 rewards_
    = IDistributor(distributor).getDistributedRewards(rewardPoolIndex_,
    address(this)) -
            rewardPoolsProtocolDetails[rewardPoolIndex_].distributedRewards;
```

Hence, `stake` will be bricked.

### Recommendations:

Consider adding a re-entrant guard to `stake`.

**Morpheus:** Acknowledged.

## [M-2] Using the same token in multiple `DepositPools` causes incorrect yield accounting

| SEVERITY: Medium | IMPACT: Medium |
| --- | --- |
| STATUS: Resolved | LIKELIHOOD: Low |

### Target

- Distributor.sol#L358
- Distributor.sol#L172-L225

### Description:

When `distributeRewards` is called and yield of each pool is calculated, it uses token balance inside the `Distributor` to determine the earned yield.

- Distributor.sol#L358

```
    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)
        );
        rewardPoolLastCalculatedTimestamp[rewardPoolIndex_]
    = 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_;

            return;
        }

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

        //// 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_;
        }
// ...
}
```

This means if there are duplicate `depositPool.token` or `depositPool.aToken` entries, the pool's yield will be miscalculated, leading to incorrect reward distribution across pools.

This scenario is possible because `addDepositPool` doesn't validate whether `aToken` or `token_` is already used by another pool.

- Distributor.sol#L172-L225

## Recommendations:

Consider adding validation in `addDepositPool` to ensure `aToken` and `token_` are unique for each DepositPool.

**Morpheus:** Resolved with @406ad354f30...

**Zenith:** Verified.

## 4.4 Low Risk

A total of 4 low risk findings were identified.

### [L-1] Chainlink may return stale prices leading to incorrect token valuation

| SEVERITY: Low | IMPACT: Low |
| --- | --- |
| STATUS: Resolved | LIKELIHOOD: Low |

#### Target

- ChainLinkDataConsumer.sol#L75-L89

#### Description:

When `getChainLinkDataFeedLatestAnswer` is called to query token prices, it doesn't validate the freshness of the returned data. This could result in the protocol using outdated prices.

```
function
    getChainLinkDataFeedLatestAnswer(bytes32 pathId_)
    external view returns (uint256) {
    address[] memory dataFeeds_ = dataFeeds[pathId_];

    uint256 res_ = 0;
    uint8 baseDecimals_ = 0;
     for
    (uint256 i = 0; i < dataFeeds_.length; i++) {
        AggregatorV3Interface aggregator_
    = AggregatorV3Interface(dataFeeds_[i]);

        try aggregator_.latestRoundData()
        returns
        (uint80, int256 answer_, uint256, uint256, uint80) {
            if (answer_ <= 0) {
                return 0;
            }

            if (res_ == 0) {
```

```
                res_ = uint256(answer_);
                baseDecimals_ = aggregator_.decimals();
            } else {
                res_ = (res_ * uint256(answer_))
    / (10 ** aggregator_.decimals());
            }
        } catch {
            return 0;
        }
    }

    return res_.convert(baseDecimals_, 18);
}
```

## Recommendations:

It's recommended to implement a freshness check by validating the `updatedAt` timestamp returned by the oracle against the feed's heartbeat, return 0 if the price is stale.

**Morpheus:** Resolved with @406ad354f30...

**Zenith:** Verified.

## [L-2] Updating `rewardPoolLastCalculatedTimestamp` with zero rewards enables griefing attack

| SEVERITY: Low | IMPACT: Low |
| --- | --- |
| STATUS: Resolved | LIKELIHOOD: Low |

### Target

- Distributor.sol#L322-L323

### Description:

When `distributeRewards` is called and rewards is 0, it still updates `rewardPoolLastCalculatedTimestamp`.

```
    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)
        );
>>>     rewardPoolLastCalculatedTimestamp[rewardPoolIndex_]
    = uint128(block.timestamp);
>>>     if (rewards_ == 0) return;
        // ...
}
```

This can cause issue, `rewards` can be 0 in certain scenario where the interval between `lastCalculatedTimestamp_` and current `block.timestamp` is small enough causing

calculated reward to be 0. The likelihood increases with tokens that have low decimals.

- LinearDistributionIntervalDecrease.sol#L49-L54

```
    function getPeriodReward(
        uint256 initialAmount_,
        uint256 decreaseAmount_,
        uint128 payoutStart_,
        uint128 interval_,
        uint128 startTime_,
        uint128 endTime_
    ) external pure returns (uint256) {
        if (interval_ == 0) {
            return 0;
        }

        // 'startTime_' can't be less than 'payoutStart_'
        if (startTime_ < payoutStart_) {
            startTime_ = payoutStart_;
        }

        uint128 maxEndTime_ = _calculateMaxEndTime(payoutStart_, interval_,
    initialAmount_, decreaseAmount_);

        if (endTime_ > maxEndTime_) {
            endTime_ = maxEndTime_;
        }

        // Return 0 when calculation 'startTime_' is bigger then
    'endTime_'...
        if (startTime_ >= endTime_) {
            return 0;
        }

        // Calculate interval that less then 'interval_' range
        uint256 timePassedBefore_ = startTime_ - payoutStart_;
        if ((timePassedBefore_ / interval_) == ((endTime_ - payoutStart_)
    / interval_)) {
            uint256 intervalsPassed_ = timePassedBefore_ / interval_;
            uint256 intervalFullReward_ = initialAmount_ - intervalsPassed_
    * decreaseAmount_;

>>>         return (intervalFullReward_ * (endTime_ - startTime_))
    / interval_;
        }
    // ...
}
```

### Recommendations:

Consider to update `rewardPoolLastCalculatedTimestamp` after the `rewards` early return check.

**Morpheus:** Resolved with @406ad354f30...

**Zenith:** Verified.

## [L-3] Migration will fail if the pool has no yield

| SEVERITY: Low | IMPACT: Low |
| --- | --- |
| STATUS: Acknowledged | LIKELIHOOD: Low |

### Target

- DepositPool.sol#L152

### Description:

The migration requires `remainder_` to be greater than 0, which can cause issues if the yield was just bridged / recently collected.

- DepositPool.sol#L152

```
    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_);
    }
```

### Recommendations:

Remove the strict non-zero requirement for `remainder_`, and transfer the yield to the distributor if it is greater than 0.

**Morpheus:** Yield is necessary so that stakers do not lose their rewards during the migration process. The solution is to plan the migration at least one day after collecting the yield using the Morpheus protocol.

**Zenith:** Acknowledged.

## [L-4] Self-referring is possible and can potentially be abused by users

| SEVERITY: Low | IMPACT: Low |
| --- | --- |
| STATUS: Acknowledged | LIKELIHOOD: Low |

### Target

- DepositPool.sol#L403-L411
- DepositPool.sol#L758-L762

### Description:

When users stake tokens in the `DepositPool` and provide a non-zero `referrer_`, they receive an extra 1% multiplier, and the `referrer_` gets a deposit bonus, allowing them to receive a portion of the rewards distributed to the pool.

- ReferrerLib.sol#L11-L19

```
uint256 constant REFERRAL_MULTIPLIER = (PRECISION * 101) / 100; // 1%
    referral bonus

function
    getReferralMultiplier(address referrer_) external pure returns (uint256)
    {
    if (referrer_ == address(0)) {
        return PRECISION;
    }

    return REFERRAL_MULTIPLIER;
}
```

- ReferrerLib.sol#L30-L45

```
function applyReferrerTier(
    IReferrer.ReferrerData
        storage referrerData,
    IReferrer.ReferrerTier[]
        storage referrerTiers,
    uint256 oldAmount_,
```

```
    uint256 newAmount_,
    uint256 currentPoolRate_
) external {
    uint256 newAmountStaked_
     = referrerData.amountStaked + newAmount_ - oldAmount_;
    uint256 multiplier_
     = _getReferrerMultiplier(referrerTiers, newAmountStaked_);
    uint256 newVirtualAmountStaked_ = (newAmountStaked_ * multiplier_)
    / PRECISION;

    referrerData.pendingRewards = getCurrentReferrerReward(referrerData,
    currentPoolRate_);
    referrerData.rate = currentPoolRate_;
    referrerData.amountStaked =
         newAmountStaked_;
    referrerData.virtualAmountStaked = newVirtualAmountStaked_;
}
```

This can be abused, users can set their own address as `referrer_` to gain an extra 1% multiplier and additional `virtualAmountStaked`. For example, if the referrer multiplier threshold is 5% and the user reaches it, they would receive an extra 6% simply by referring themselves, compared to another user who doesn't provide a `referrer_`.

## Recommendations:

Whitelist the referrer address, if the provided referrer is not whitelisted, revert the operation.

**Morpheus:** Outside the scope of the current task. Could be a suggestion for future updates.

**Zenith:** Acknowledged.

## 4.5 Informational

A total of 2 informational findings were identified.

### [I-1] Calculation method can be improved to eliminate any chance of overflow

| SEVERITY: Informational | IMPACT: Informational |
| --- | --- |
| STATUS: Resolved | LIKELIHOOD: Low |

#### Target

- RewardPool.sol

#### Description:

In `RewardPool.sol`, there is a function `getPeriodRewards` which calls the `getPeriodReward` function of the `LinearDistributionIntervalDecrease` library.

It then makes a call to `_calculateFullPeriodReward`. In the last line of that function there is a math calculation:

```
return initialReward_ * ip_ - (decreaseAmount_ * (ip_ * (ip_ - 1))) / 2;
```

We can see that `(ip_ * (ip_ - 1))` is multiplied by `decreaseAmount_` first before being divided by 2.

This is generally done to avoid rounding troubles **but** in this case we can guarantee that `(ip_ * (ip_ - 1))` is **always** an even number and hence can just be divided by 2 immediately.

- This is because `ip` and `ip - 1` are consecutive numbers, hence either is definitely even.

This is ensure that as best as possible, overflow can be prevented even if `decreaseAmount` is a large number.

#### Recommendations:

Move the `/ 2` inside since it is guaranteed that `(ip_ * (ip_ - 1))` will be even.

```
return initialReward_ * ip_ - (decreaseAmount_ * (ip_ * (ip_ - 1))) / 2;
return initialReward_ * ip_ - decreaseAmount_ * (ip_ * (ip_ - 1) / 2);
```

**Morpheus:** Resolved with @406ad354f300...

**Zenith:** Verified.

## [I-2] Missing claim for Aave incentives inside `Distributor` leads to unclaimed rewards

| SEVERITY: Informational | IMPACT: Informational |
| --- | --- |
| STATUS: Resolved | LIKELIHOOD: Low |

### Target

- Distributor.sol

### Description:

Aave provides Incentives (e.g., incentives to the supply or borrow side of Aave liquidity pools, seeing here: https://aave.com/docs/primitives/incentives) to users who supply assets to the protocol. These incentives are typically distributed in the form of additional tokens (e.g., AAVE or other governance tokens) and can be claimed by users who interact with Aave's incentive mechanisms.

However, in the current implementation of the `Distributor` contract, there is no functionality to claim these incentives. This is a missing feature that could prevent claim the full benefits of supplying assets to Aave.

### Recommendations:

Introduce a function that allows claiming rewards from Aave's `RewardsController`.

**Morpheus:** Resolved with @406ad354f300...

**Zenith:** Verified.