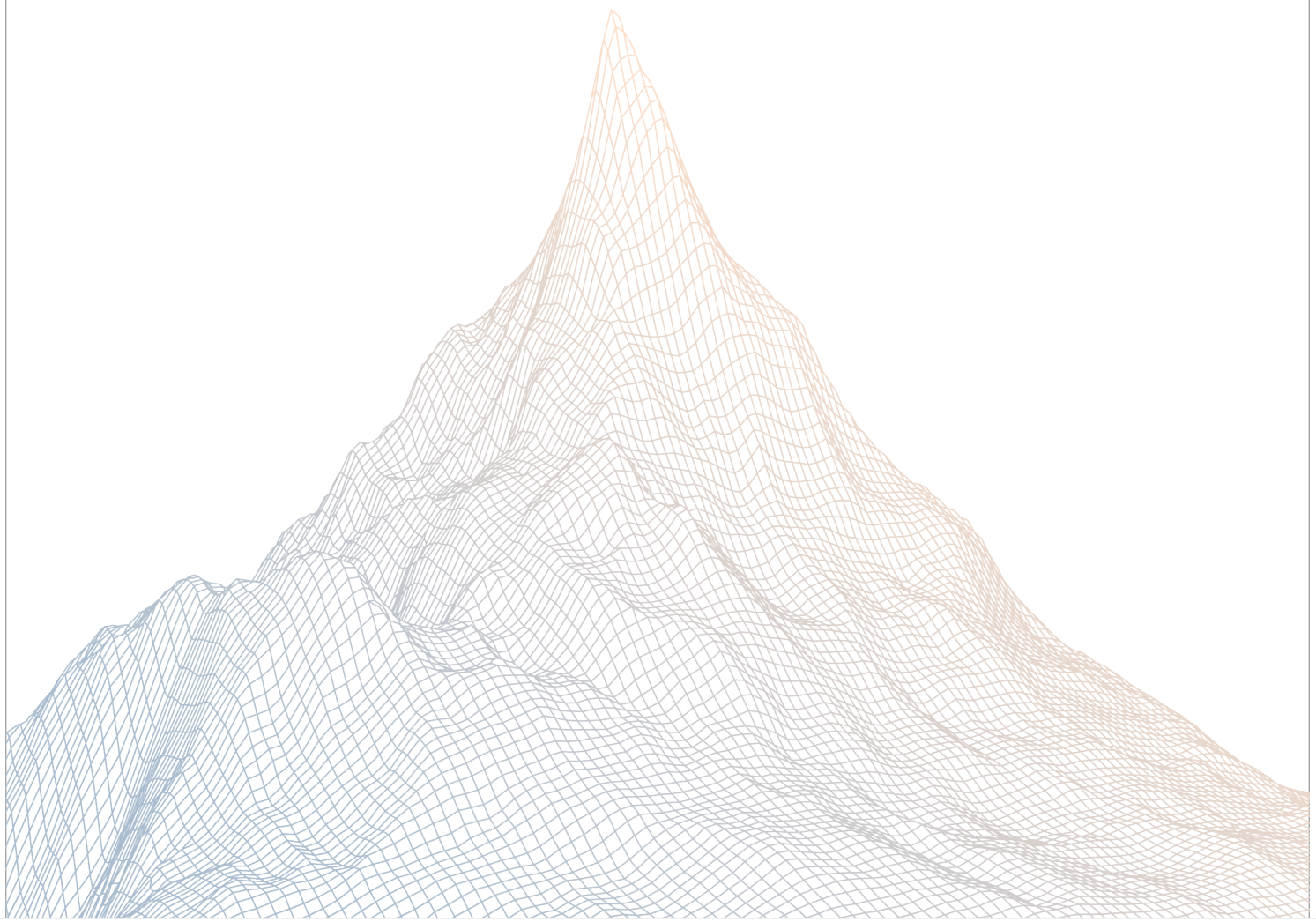
Zenith

Lindy

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: June 24th to June 26th, 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 Lindy

In alignment with the ethos of the space, Sandclock leverages the blockchain's unique properties of transparency, trustlessness, and composability to create a state of the art and the world's first trustless wealth management platform to supercharge your capital's programmability in an accessible manner. Sandclock's ambition is to one day be the masses' gateway to the world of Decentralized Finance.

2.2 Scope

The engagement involved a review of the following targets:

Target	audit-mono
Repository	https://github.com/lindy-labs/audit-mono
Commit Hash	`415c65cc64d08e2799965896ab2df7ee0a7c7f91`
Files	`sc_sandy/src/*` `sandclock-loop/src/*`

2.3 Audit Timeline

June 24th, 2025	Audit start
June 26th, 2025	Audit end
July 26th, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	1
High Risk	5
Medium Risk	2
Low Risk	6
Informational	1
Total Issues	**15**

3

Findings Summary

ID	Description	Status
C-1	Claimers can steal more sandy tokens due to incorrect calculation	Resolved
H-1	Users may avoid locking WETH in SandyLockDrop due to the fixed price	Resolved
H-2	eLoop repay and supply process incorrect amounts of assets and debt.	Resolved
H-3	Lack of position threshold in aLoop and eLoop could lead to liquidation.	Acknowledged
H-4	Users can avoid the aLoop/eLoop performance fee	Acknowledged
H-5	aLoop's EMode is misconfigured for Mainnet	Resolved
M-1	eLoop and aLoop repay and supply do not accrue a performance fee	Acknowledged
M-2	aLoop and eLoop performance fee share calculation is incorrect	Resolved
L-1	Consider encoding chainid in the hash if distributor is used in multiple chains	Acknowledged
L-2	hasClaimed check inside claimSandy could prevents multiple valid claims	Resolved
L-3	Sandy token rewards claiming does not account for usdc depeg senario	Resolved
L-4	Hardcoded Aave V3 pool address may cause issue in future upgrade	Acknowledged
L-5	Missing claim for Aave incentives leads to unclaimed rewards	Acknowledged
L-6	Lack of slippage protection for aLoop/eLoop repay and supply operations	Acknowledged
I-1	Denominator should be rounded up for consistency	Resolved

4

Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] Claimers can steal more sandy tokens due to incorrect calculation

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

Target

- SandyLockDrop.sol-L232
- SandyLockDrop.sol-L294

Description:

In `SandyLockDrop.sol`, when the user is trying to claim their Sandy rewards using `claimSandyRewards`, the code calculates the reward token amounts using `calculateSandyReward`.

But there is an error in the calculation and the `totalAssetValue` to divide by does not include all the assets.

`claimSandyRewards` outsources the calculation to `calculateSandyReward` which calculates the denominator like this:

```
// Similarly, combine total USD values
uint256 totalAssetValue
    = lockedAmountsAdjusted[Constants.WETH].mulWadDown(WETH_USD_PRICE)
    + lockedAmountsAdjusted[Constants.USDC] * 1e12;
```

However if we look at `_lockToken`, in the `else if` branch:

```
} else if (_token == Constants.SCWETHV2 || _token
      == Constants.SCUSDCV2) {
      uint256 underlyingAmount
      = IERC4626(_token).convertToAssets(_amount);
```

```
        user.lockedToken = IERC4626(_token).asset();
        user.lockedAssets += underlyingAmount;
        user.lockedShares += _amount;

->      lockedAmountsAdjusted[_token]
        += underlyingAmount.mulWadDown(_vestingMultiplier);

        IERC4626(_token).transferFrom(msg.sender, address(this), _amount);
    }
```

Hence, users contributing USDC through the USDC vault shares for example, gets added to `lockedAmountsAdjusted[vault_address]` instead of `lockedAmountsAdjusted[Constants.USDC]`.

So consider the following senario:

1. Alice contributes \$1000 USDC.
2. Bob sees that and contributes \$1000 USDC through USDC vault shares.
3. After the deadline is over and claiming begins, Bob frontruns and be the first to claim.
4. Since the total does not include everything properly as described above, `totalAssetValue` = `lockedAmountsAdjusted[Constants.USDC]` = \$1000
5. Bob is calculated as having contributed 100% of the asset value.
6. Bob steals more Sandy tokens
7. Now, when Alice claims, her transaction gets reverted as sandy token balance in the contract is now insufficient

Hence, Bob manages to steal more Sandy tokens.

Recommendations:

In `_lockToken`:

```
lockedAmountsAdjusted[_token]
+= underlyingAmount.mulWadDown(_vestingMultiplier);
lockedAmountsAdjusted[IERC4626(_token).asset()]
+= underlyingAmount.mulWadDown(_vestingMultiplier);
```

Or alternatively go to L233 and add `lockedAmountsAdjusted[vault_addresses]`.

Lindy: Resolved with @ec23113...

Zenith: Verified

4.2 High Risk

A total of 5 high risk findings were identified.

[H-1] Users may avoid locking `WETH` in `SandyLockDrop` due to the fixed price

`SEVERITY`: High	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: High

Target

- SandyLockDrop.sol#L225

Description:

In `SandyLockDrop`, the Sandy reward is calculated based on the total USDC value staked (`assetContribution`) and the total staked Sandy (`sandyContribution`).

```
    function calculateSandyReward(address _user)
    public view returns (uint256) {
        // Cache user data to reduce storage reads
        UserData memory user = users[_user];

        // If user has already claimed rewards or has no SANDY, return 0 early
        if (user.sandyRewardsClaimed || user.lockedSandy == 0) return 0;

        // These multiplications can be done once and reused
>>>     uint256 userWETHInUSD = user.lockedToken == Constants.WETH ?
    user.lockedAssets.mulWadDown(WETH_USD_PRICE) : 0;
        uint256 userUSDCInUSD = user.lockedToken == Constants.USDC ?
    user.lockedAssets * 1e12 : 0; // Convert to 18 decimals

        // Combine user's total USD value with a single multiplication by the
    multiplier
        uint256 userAssetValue = (userWETHInUSD
    + userUSDCInUSD).mulWadDown(user.vestingMultiplier);

        // Similarly, combine total USD values
```

```
>>>     uint256 totalAssetValue
    = lockedAmountsAdjusted[Constants.WETH].mulWadDown(WETH_USD_PRICE)
            + lockedAmountsAdjusted[Constants.USDC] * 1e12;

        // Calculate contribution
        uint256 assetContribution
    = userAssetValue.divWadDown(totalAssetValue);
        uint256 sandyContribution =
            user.lockedSandy.mulDivDown(user.vestingMultiplier,
    lockedAmountsAdjusted[address(sandyToken)]);

        // Calculate final reward with a single multiplication
        return (assetContribution.mulWadDown(WEIGHT_ASSETS)
    + sandyContribution.mulWadDown(WEIGHT_SANDY)).mulWadDown(
            rewardSandySupply
        );
    }
```

The issue is that for users staking WETH in the contract, its value is converted to USDC using a fixed `WETH_USD_PRICE` (1800e18). This may discourage users from staking WETH (At the time of reporting this issue, ETH is priced at approximately 2400 USDC). This is especially concerning for users who want to lock assets for a longer period.

Recommendations:

Consider creating a separate allocation of `rewardSandySupply` for WETH and USDC lockers, and calculate each allocation for users based on the `lockedAmountsAdjusted` of the corresponding asset.

Lindy: Resolved with @7cbc92f...

Zenith: Verified.

[H-2] `eLoop` `repay` and `supply` process incorrect amounts of assets and debt.

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

Target

- eLoop.sol#L65
- eLoop.sol#L71

Description:

When `repay` or `supply` is called, it converts the vault's shares to debt and vice versa.

However, It can be seen that inside `repay`, it incorrectly provide underlying collateral amount to the vault's share `transfer` operation.

```
    // decreases leverage, any amount, likely to net a loss
    function repay(uint256 amount) public {
        // set the max repayable to the current debt
        uint256 max = dToken.debtOf(address(this));
        amount = amount > max ? max : amount;
        debt.transferFrom(msg.sender, address(this), amount);
        debt.approve(address(dToken), amount);
        dToken.repay(amount, address(this));
>>>     asset.transfer(msg.sender, toCollateral(amount));
    }

The similar case inside the `supply` operation, it incorrectly provides the
    vault's share amount to `toDebt` instead of the underlying asset amount.

    // increase the leverage, any amount, likely to net a profit
    function supply(uint256 amount) public {
        asset.transferFrom(msg.sender, address(this), amount);
>>>     amount = toDebt(amount);
        dToken.borrow(amount, msg.sender);
    }
```

This will result in these operations processing incorrect amounts, which is especially dangerous for `repay`, as it allow users to receive more vault shares than they should.

Recommendations:

use the correct shares / underlying asset amount :

```
    // decreases leverage, any amount, likely to net a loss
    function repay(uint256 amount) public {
        // set the max repayable to the current debt
        uint256 max = dToken.debtOf(address(this));
        amount = amount > max ? max : amount;
        debt.transferFrom(msg.sender, address(this), amount);
        debt.approve(address(dToken), amount);
        dToken.repay(amount, address(this));
        asset.transfer(msg.sender, toCollateral(amount));
        uint256 receivedShares = ERC4626(address(asset)).convertToShares(
            toCollateral(amount));
        asset.transfer(msg.sender, receivedShares);
    }

The similar case inside the `supply` operation, it incorrectly provides the
    vault's share amount to `toDebt` instead of the underlying asset amount.

    // increase the leverage, any amount, likely to net a profit
    function supply(uint256 amount) public {
        asset.transferFrom(msg.sender, address(this), amount);
        amount = toDebt(amount);
        amount = toDebt(ERC4626(address(asset)).convertToAssets(amount));
        dToken.borrow(amount, msg.sender);
    }
```

Lindy: Resolved with @f465a4a...

Zenith: Verified.

[H-3] Lack of position threshold in `aLoop` and `eLoop` could lead to liquidation.

SEVERITY: High	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- aLoop.sol#L65-L81
- eLoop.sol#L58-L73

Description:

When users call `supply` or `repay`, there is no check or protection for the position inside the `aLoop` and `eLoop`, as long as passing Aave's or Euler's health factor check, the operation will success.

```
// decreases leverage, any amount, likely to net a loss
function repay(uint256 amount) public {
    // set the max repayable to the current debt
    uint256 max = dToken.balanceOf(address(this));
    amount = amount > max ? max : amount;
    debt.transferFrom(msg.sender, address(this), amount);
    debt.approve(address(pool), amount);
    pool.repay(address(debt), amount, 2, address(this));
    pool.withdraw(collateral, toCollateral(amount), msg.sender);
}

// increase the leverage, any amount, likely to net a profit
function supply(uint256 amount) public {
    asset.transferFrom(msg.sender, address(this), amount);
    amount = toDebt(amount);
    pool.borrow(address(debt), amount, 2, 0, address(this));
    debt.transfer(msg.sender, amount);
}
```

This is unsafe, as users can freely `redeem` or `withdraw` collateral from the vault, bringing the health factor closer to the liquidation threshold and making the system prone to liquidation.

Recommendations:

Consider configuring a maximum position or debt-to-collateral threshold (as a safety margin from the actual health factor) within `aLoop` and `eLoop`, and perform the threshold check it during `supply` or `repay` operations.

Lindy: Acknowledged

[H-4] Users can avoid the aLoop/eLoop performance fee

SEVERITY: High	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: High

Target

- aLoop.sol#L53-L62
- eLoop.sol#L46-L55

Description:

When `beforeWithdraw` is called, the `aLoop/eLoop` profit is calculated. During this operation, the `profit` is determined, and shares are minted to the vault based on the `performanceFee` percentage of that `profit`.

```
    function beforeWithdraw(uint256 assets, uint256) internal override {
        // see if profits and then collect performance fee
        if (totalAssets() > cachedTotalAssets) {
>>>         uint256 profit = totalAssets() - cachedTotalAssets;
>>>         _mint(address(this),
    convertToShares(profit.mulWadDown(performanceFee)));
        }

        // otherwise it is "loss"
        cachedTotalAssets = totalAssets() - assets;
    }
```

However, when a user calls `redeem` or `withdraw`, the amount of assets they receive is calculated using the total supply before the fee is minted.

```
    function redeem(
        uint256 shares,
        address receiver,
        address owner
    ) public virtual returns (uint256 assets) {
        if (msg.sender ≠ owner) {
            uint256 allowed = allowance[owner][msg.sender]; // Saves gas for
    limited approvals.
```

```
            if (allowed ≠ type(uint256).max) allowance[owner][msg.sender]
    = allowed - shares;
        }

        // Check for rounding error since we round down in previewRedeem.
>>>     require((assets = previewRedeem(shares)) ≠ 0, "ZERO_ASSETS");

>>>     beforeWithdraw(assets, shares);

        _burn(owner, shares);

        emit Withdraw(msg.sender, receiver, owner, assets, shares);

        asset.safeTransfer(receiver, assets);
    }
```

This means that the caller of `redeem` or `withdraw` at the time the fee is minted will receive more assets than they should, as they receive the full profit shares before the fee is deducted. If the user is the only one in the vault, the performance fee will be zero.

PoC :

Add the following test to `Loop.t.sol` :

```
function test_redeem_race_condition() public {
    uint256 amount = 1e18;
    deal(address(underlying), alice, amount);
    deal(address(underlying), bob, amount);
    // prepare for simulating profit
    deal(address(underlying), address(this), amount);
    underlying.approve(address(vault.pool()), amount);
    pool.supply(address(underlying), underlying.balanceOf(address(this)),
    address(this), 0);

    // alice deposit
    vm.startPrank(alice);
    underlying.approve(address(vault.pool()), amount);
    pool.supply(address(underlying), underlying.balanceOf(alice), alice, 0);

    asset.approve(address(vault), amount);
    vault.deposit(amount, alice);
    vm.stopPrank();

    // bob deposit same amount
    vm.startPrank(bob);
    underlying.approve(address(vault.pool()), amount);
```

```
    pool.supply(address(underlying), underlying.balanceOf(bob), bob, 0);

    asset.approve(address(vault), amount);
    vault.deposit(amount, bob);
    vm.stopPrank();

    // simulate profit
    asset.transfer(address(vault), amount);

    vm.startPrank(bob);
    vault.redeem(vault.balanceOf(address(bob)), address(bob), address(bob));
    vm.stopPrank();

    vm.startPrank(alice);
    vault.redeem(vault.balanceOf(address(alice)), address(alice),
    address(alice));
    vm.stopPrank();

    assertGt(asset.balanceOf(bob), asset.balanceOf(alice));
}
```

Recommendations:

Consider deducting the `performanceFee` before calculating assets or shares.

Lindy: Acknowledged.

[H-5] aLoop's EMode is misconfigured for Mainnet

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

Target

- aLoop.sol#L35

Description:

Inside the aLoop constructor, the vault's EMode category is hardcoded to 2.

```
    constructor(address _aToken, address _dToken) ERC4626(ERC20(_aToken),
    "", "") {
        // ...

        // set e-mode on aave
>>>     pool.setUserEMode(2);

        approve(msg.sender, type(uint256).max);
    }
```

This is correct for the Aave V3's Arbitrum chain configuration, but incorrect on Aave V3's Mainnet configuration. In Mainnet, CategoryId 2 is for sUSDe Stablecoins.

Source : etherscan address

Wrongly configured EMode will cause the vault to always revert when `supply` .

Run 'V3Test:test_supply' and see the error :

```
forge test --match-test test_supply --match-contract V3Test -vvv


output :

[Revert] 100
```

In the Aave codebase, 100 is defined as the `NOT_BORROWABLE_IN_EMODE` error code.

Recommendations:

Allow the deployer to provide the EMode category ID to the constructor. On mainnet, the correct category ID for the ETH-Correlated category is 1.

Lindy: Resolved with @cde4b41...

Zenith: Verified

4.3 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] `eLoop` and `aLoop` `repay` and `supply` do not accrue a performance fee

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- aLoop.sol#L65-L81
- eLoop.sol#L58-L73

Description:

`repay` and `supply` operations in `eLoop` and `aLoop` change the debt leverage within the vaults and can potentially affect the vault's profitability or loss.

```
// decreases leverage, any amount, likely to net a loss
function repay(uint256 amount) public {
    // set the max repayable to the current debt
    uint256 max = dToken.balanceOf(address(this));
    amount = amount > max ? max : amount;
    debt.transferFrom(msg.sender, address(this), amount);
    debt.approve(address(pool), amount);
    pool.repay(address(debt), amount, 2, address(this));
    pool.withdraw(collateral, toCollateral(amount), msg.sender);
}

// increase the leverage, any amount, likely to net a profit
function supply(uint256 amount) public {
    asset.transferFrom(msg.sender, address(this), amount);
    amount = toDebt(amount);
    pool.borrow(address(debt), amount, 2, 0, address(this));
    debt.transfer(msg.sender, amount);
}
```

However, when the debt leverage is changed through these operations, no performance fee is taken. Consider a scenario where `totalAssets()` is greater than `cachedTotalAssets` (`profit` > 0), and then a `supply` or `repay` operation is performed. After some time, `totalAssets()` is no longer greater than `cachedTotalAssets`, the opportunity to charge the performance fee is lost.

Recommendations:

accrue performance fee and update `cachedTotalAssets` when `supply` or `repay` is performed.

Lindy: Acknowledged.

[M-2] aLoop and eLoop performance fee share calculation is incorrect

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- eLoop.sol#L49-L50
- aLoop.sol#L56-L57

Description:

When calculating performance fee inside `beforeWithdraw`, it will calculate shares using `convertToShares`, which will include the performance fee amount inside the `totalAssets()`.

```
    function beforeWithdraw(uint256 assets, uint256) internal override {
        // see if profits and then collect performance fee
        if (totalAssets() > cachedTotalAssets) {
>>>         uint256 profit = totalAssets() - cachedTotalAssets;
>>>         _mint(address(this),
    convertToShares(profit.mulWadDown(performanceFee)));
        }

        // otherwise it is "loss"
        cachedTotalAssets = totalAssets() - assets;
    }
```

```
function convertToShares(uint256 assets)
    public view virtual returns (uint256) {
    uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
    non-zero.

    return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
}
```

This will cause the minted shares to be worth less than the calculated performance fee collateral amount.

Recommendations:

Instead of use `convertToShares`, consider to calculate it with the following calculation :

```
_mint(address(this), convertToShares(profit.mulWadDown(performanceFee)));
uint256 performanceFeeAmount = profit.mulWadDown(performanceFee);
uint256 sharesMinted = performanceFeeAmount.mulDivDown(totalSupply,
    totalAssets() - performanceFeeAmount);
_mint(address(this), sharesMinted);
```

Lindy: Resolved with @0d33a6b...

Zenith: Verified

4.4 Low Risk

A total of 6 low risk findings were identified.

[L-1] Consider encoding chainid in the hash if distributor is used in multiple chains

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- SandyDistributor.sol-L91

Description:

```
function claimSandy(uint256 _amount, bytes32[] calldata _proof) external {
    ....

    // Verify merkle proof using OpenZeppelin's library
    bytes32 leaf = keccak256(abi.encodePacked(msg.sender, _amount));
    if (!MerkleProof.verify(_proof, merkleRoot, leaf))
    revert InvalidProof();

    ....
}
```

Since the sandy token is a multi-chain token, in the senario where `SandyDistributor.sol` is deployed on multiple chains, it is better to encode chainid in the leaf hash to prevent cross-chain replays.

Sometimes projects use an overarching main merkle root across the same contract in different chains, but of course designate off-chain which chain's distributor the user should be claiming from.

In such senarios, encoding the `chainid` will ensure that the user only can claim from that one contract in that one designated chain

Recommendations:

Encode `chainid` to prevent cross-chain replays if sandyDistributor is intended to be deployed on other chains in the future.

```
bytes32 leaf =
keccak256(abi.encodePacked(msg.sender, _amount));
bytes32 leaf =
keccak256(abi.encodePacked(msg.sender, _amount, block.chainid));
 if (!MerkleProof.verify(_proof, merkleRoot, leaf))
   revert InvalidProof();
```

Lindy: Acknowledged.

[L-2] `hasClaimed` check inside `claimSandy` could prevents multiple valid claims

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- SandyDistributor.sol#L88-L97

Description:

When users claim Sandy in `SandyDistributor`, `hasClaimed` will be set to true for the `msg.sender`

```
    function claimSandy(uint256 _amount, bytes32[] calldata _proof)
    external {
        if (_amount == 0) revert InvalidAmount();
>>>     if (hasClaimed[msg.sender]) revert AlreadyClaimed();

        // Verify merkle proof using OpenZeppelin's library
        bytes32 leaf = keccak256(abi.encodePacked(msg.sender, _amount));
        if (!MerkleProof.verify(_proof, merkleRoot, leaf))
    revert InvalidProof();

        uint256 sandyBalance = _getSandyBalance();
        if (sandyBalance < _amount) revert InsufficientBalance(_amount,
    sandyBalance);

>>>     hasClaimed[msg.sender] = true;

        // unchecked for gas savings
        unchecked {
            totalSandyDistributed += _amount;
        }

        SANDY.safeTransfer(msg.sender, _amount);

        emit SandyClaimed(msg.sender, _amount);
    }
```

If user has another allocation amounts (different leaf) in the Merkle root, they cannot claim it.

Recommendations:

Set `hasClaimed` based on the `leaf` instead of `msg.sender`.

Lindy: Resolved with @ec23113...

Zenith: Verified.

[L-3] Sandy token rewards claiming does not account for usdc depeg senario

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- SandyLockDrop.sol-L226

Description:

When the user claims their Sandy token rewards, the amount of tokens they receive is calculated using the function `calculateSandyReward`.

The function `calculateSandyReward` takes into account the user deposited tokens over the total tokens deposited.

```
function calculateSandyReward(address _user) public view returns (uint256) {
    ....
    uint256 userUSDCInUSD = user.lockedToken == Constants.USDC ?
    user.lockedAssets * 1e12 : 0;
    ....
}
```

When calculating the user contributed `USDC` value, it is assumed to be always `1:1`.

However, in an extreme senario, USDC could possibly experience a big depeg (and perhaps even remain depegged for a sizeable amount of time) which means the current code might end up overvaluing users' USDC contributions.

- This will shortchange users who contributed `WETH` as by right they are now suppose to have a larger share of the rewards.

Recommendations:

Consider adding a hardcoded `PRICE` multiplier variable which will be set to `1e18`, and only in extreme situations where the admin can change it accordingly.

- Note that the existing `WEIGHT_ASSETS` and `WEIGHT_SANDY` does not solve this issue as that is meant to adjust the weightage of assets vs sandy contributions, but the weight shift

needed here is between `WETH` and `USDC`.

Lindy: Resolved with @7cbc92f...

Zenith: Verified

[L-4] Hardcoded Aave V3 pool address may cause issue in future upgrade

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- aLoop.sol#L23

Description:

According to the Aave documentation, whenever the `Pool` contract is needed, the `PoolAddressProvider` contract should be queried each time to obtain the current pool address.

However, the `pool` inside `aLoop` is currently immutable and set only once in the constructor.

Recommendations:

A recommended approach is to use `addressProviderAddress.getPool()` to reference pool instead of hardcoding.

Lindy: Acknowledged.

[L-5] Missing claim for Aave incentives leads to unclaimed rewards

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- aLoop.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 `aLoop` 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`.

Lindy: Acknowledged.

[L-6] Lack of slippage protection for `aLoop`/`eLoop` `repay` and `supply` operations

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- eLoop.sol#L58-L73
- aLoop.sol#L65-L81

Description:

`aLoop` / `eLoop` `repay` and `supply` operations convert the debt amount to collateral and vice versa based on the ratio or the oracle price.

```
function repay(uint256 amount) public {
    // set the max repayable to the current debt
    uint256 max = dToken.balanceOf(address(this));
    amount = amount > max ? max : amount;
    debt.transferFrom(msg.sender, address(this), amount);
    debt.approve(address(pool), amount);
    pool.repay(address(debt), amount, 2, address(this));
    pool.withdraw(collateral, toCollateral(amount), msg.sender);
}

// increase the leverage, any amount, likely to net a profit
function supply(uint256 amount) public {
    asset.transferFrom(msg.sender, address(this), amount);
    amount = toDebt(amount);
    pool.borrow(address(debt), amount, 2, 0, address(this));
    debt.transfer(msg.sender, amount);
}
```

This means users may receive an undesired amount of collateral or debt when the request is executed, due to changes in price or ratio.

Recommendations:

Consider providing a minimum amount received parameter in the `supply` and `repay` functions.

Lindy: Acknowledged.

4.5 Informational

A total of 1 informational findings were identified.

[I-1] Denominator should be rounded up for consistency

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

Target

- SandyLockDrop.sol-L232

Description:

In `SandyLockDrop.sol`, when the user is claiming their sandy rewards, the protocol's intention is to round down in the calculation of `numerator / denominator`.

It is accurate (and intuitive) to round down to prevent the amount distributed from exceeding the contract's sandy balance.

If the overall intention is to round down, then the denominator should be rounded up. (so that the final result is rounded down)

```
function calculateSandyReward(address _user) ... {
    ....
    ....
    uint256 totalAssetValue
    = lockedAmountsAdjusted[Constants.WETH].mulWadDown(WETH_USD_PRICE)
        + lockedAmountsAdjusted[Constants.USDC] * 1e12;
    ....
}
```

However we can see that in the code, `totalAssetValue` which is the denominator, is rounded down due to `lockedAmountsAdjusted[Constants.WETH].mulWadDown`.

Recommendations:

For consistency, round **up** the denominator (so that the final value is rounded down).

Change `mulWadDown` to `mulWadUp`:

```
function calculateSandyReward(address _user) ... {
    ....
   uint256 totalAssetValue =
   lockedAmountsAdjusted[Constants.WETH].mulWadDown(WETH_USD_PRICE)
        + lockedAmountsAdjusted[Constants.USDC] * 1e12;
   uint256 totalAssetValue =
   lockedAmountsAdjusted[Constants.WETH].mulWadUp(WETH_USD_PRICE)
        + lockedAmountsAdjusted[Constants.USDC] * 1e12;
    ....
}
```

Lindy: Resolved with @ec23113...

Zenith: Verified.