Zerrow Review

A smart contract assessment of Zerrow contracts was performed by Octane Security, with a focus on the security aspects of the application's implementation.

About Octane Security

Octane Security is an AI-enhanced blockchain security firm dedicated to safeguarding digital assets. We combine security research with cutting-edge machine learning models to maximize protocol security and minimize risk. Our team ranks in the top of audit contest leaderboards, captures bug bounties, and has served big-name clients such as Alchemy, Optimism, Notional Finance, Blast, and Graph Protocol in the past.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We also rely on information that is provided by the client, its affiliates and partners for vulnerability identification. We can not guarantee the absense of vulnerabilities, issues, flaws, or defects after the review. It is up to the client to decide whether to implement recommendations and we take no liability for invalid recommendations. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

Security Assessment Summary

The scope of this review was limited to files at commit: 64ddf854c807ef7843ab63bb335eae0b74252779

Scope

The following smart contract was in scope of the audit:

contracts\template\depositOrLoanCoin.sol

contracts\wA0GI\wA0GI.sol

contracts\coinFactory.sol

contracts\lendingCoreAlgorithm.sol

contracts\lendingManager.sol

contracts\lendingVaults.sol

contracts\rewardRecordMock.sol

contracts\slcoraclemock.sol

Findings

Finding Number	Finding Name
H-1	Incorrect Token Indexing In `badDebtDeduction` Causes Reverts (FIXED) 52aa26f
H-2	`licensedAssetsReset` Fails To Update Protocol State, Leading To Incorrect Interest Calculations (FIXED) 52aa26f
H-3	Incorrect Balance Tracking Enables Interest Rate Manipulation Via Direct Transfers (FIXED) 52aa26f
M-1	Inconsistency In Utilization Rate Enforcement Between Borrowing And Liquidation (FIXED) 52aa26f
L-1	Inconsistent Asset Reference In Risk Isolation Mode Lending Validation (FIXED) dbf6915
L-2	Risk Isolation Lending Limit Can Be Exceeded Due To Accrued Interest (ACKNOWLEDGED)
I-1	Unvalidated Oracle Responses May Lead To Incorrect Lending Value Calculations (CLOSED)
I-2	Interface Removal Does Not Break Loop If Found Or Revert If Not Found (FIXED) dbf6915
I-3	Confusing Error Message In `lendAsset` To Reflect Borrow Limit Exceeded (FIXED) dbf6915
I-4	Misuse Of “Lend” Instead Of “Borrow” In Protocol Logic (ACKNOWLEDGED)
I-5	`_userRIMAssetsAddress` Not Reset When Leaving Mode 1 (ACKNOWLEDGED)

Title: H-1 Incorrect Token Indexing In `badDebtDeduction` Causes Reverts

Description

In the function `badDebtDeduction`, the protocol attempts to mint the user’s deposit and loan tokens to the `badDebtCollectionAddress` and then burn the same amounts from the user’s address:

```
for (uint i = 0; i != len; i++) {
    if (depositBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
[0]).mintCoin(badDebtCollectionAddress, depositBalances[i]);
    }
    if (lendingBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
[0]).mintCoin(badDebtCollectionAddress, lendingBalances[i]);
    }
}

// Process all burns after mints
for (uint i = 0; i < len; i++) {
    if (depositBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
```

```
[0]).burnCoin(user, depositBalances[i]);
    }
    if (lendingBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
[0]).burnCoin(user, lendingBalances[i]);
    }
}
```

The problem lies in how the loan tokens are processed. When minting loan tokens to `badDebtCollectionAddress`, the code mistakenly mints **deposit tokens** instead by referencing `assetsDepositAndLend[...][0]` instead of `assetsDepositAndLend[...][1]`. As a result, the deposit token is minted twice (once for the deposit balance, and once incorrectly for the loan balance).

Similarly, when burning from the user's account, the same mistake occurs—the code tries to burn **deposit tokens** equal to the loan balance rather than burning loan tokens. This causes the transaction to revert because the user won't have enough deposit tokens to cover the burn.

In effect, the function becomes non-functional due to this incorrect handling of indices.

Recommendation

The mint and burn logic should be corrected to reference the loan token properly:

```
for (uint i = 0; i != len; i++) {
    if (depositBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
[0]).mintCoin(badDebtCollectionAddress, depositBalances[i]);
    }
    if (lendingBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
[1]).mintCoin(badDebtCollectionAddress, lendingBalances[i]);
    }
}

// Process all burns after mints
for (uint i = 0; i < len; i++) {
    if (depositBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
[0]).burnCoin(user, depositBalances[i]);
    }
    if (lendingBalances[i] > 0) {
        iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]]
[1]).burnCoin(user, lendingBalances[i]);
    }
}
```

Title: H-2 `licensedAssetsReset` Fails To Update Protocol State, Leading To Incorrect Interest

Calculations

Description

When the function `licensedAssetsReset` is invoked, it applies new configuration parameters for the specified `_asset`:

```
function licensedAssetsReset(
    address _asset,
    uint _maxLTV,
    uint _liqPenalty,
    uint _maxLendingAmountInRIM,
    uint _bestLendingRatio,
    uint _reserveFactor,
    uint8 _lendingModeNum,
    uint _homogeneousModeLTV,
    uint _bestDepositInterestRate
) public onlySetter {
    //....
}
```

The issue is that the protocol's state variables — `latestDepositCoinValue`, `latestLendingCoinValue`, `latestTimeStamp`, `latestDepositInterest`, and `latestLendingInterest` — are **not updated** when this function is called. In other words, the function should invoke `_beforeUpdate` at the start and `_assetsValueUpdate` at the end, but it currently does not.

To illustrate, suppose on **day 10** an asset is deposited, updating the protocol state so that `assetInfos[token].latestTimeStamp = day 10`. Then, on **day 15**, `licensedAssetsReset` is executed, updating the configuration of the `_asset`. At this point, one would expect the new interest rate (from the reset) to apply. However, when a deposit/withdraw/liquidate/borrow/repay action is executed on **day 30**, the function `_beforeUpdate` is triggered. This function recalculates coin values using the interest rate from **day 10**, not from **day 15**.

As a result, the recalculated value uses:

```
(day 30 - day 10) * interestRate / 365 days
```

instead of factoring in the new settings applied on day 15. This causes incorrect accrual of interest.

```
function getCoinValues(address token) public view returns(uint[2] memory
currentValue) {
    uint tempVaule = (block.timestamp - assetInfos[token].latestTimeStamp) * 1
ether / (ONE_YEAR * UPPER_SYSTEM_LIMIT);
    currentValue[0] = assetInfos[token].latestDepositCoinValue
                    + tempVaule * assetInfos[token].latestDepositInterest;
    currentValue[1] = assetInfos[token].latestLendingCoinValue
```

```
                    + tempVaule * assetInfos[token].latestLendingInterest;

        if(currentValue[0] == 0) {
            currentValue[0] = 1 ether;
        }
        if(currentValue[1] == 0) {
            currentValue[1] = 1 ether;
        }
    }
```

Recommendation

Ensure `_beforeUpdate` is executed at the beginning and `_assetsValueUpdate` at the end of `licensedAssetsReset` to properly synchronize protocol state with the new settings.

Title: H-3 Incorrect Balance Tracking Enables Interest Rate Manipulation Via Direct Transfers

Description

There is an issue in how the protocol tracks token balances, which can allow manipulation of the interest rate through direct user transfers or malicious donations into the protocol.

Consider the following scenario:

- Alice deposits 100 Token A, so 100 deposit tokens (representing Token A) are minted (assuming 1:1 price for simplicity).
- Another user then borrows 60 Token A. At this point, the protocol holds only 40 Token A, and 60 loan tokens are minted.

Now suppose Alice wants to withdraw her deposit. Since only 40 Token A are left in the protocol, she should not be able to fully withdraw. However, if any user mistakenly (or maliciously) transfers 60 Token A directly into the protocol contract, the protocol balance rises to 100 Token A. This extra balance should normally be handled through the `excessDisposal` function in the `lendingVaults` contract.

But if Alice now calls `withdrawDeposit`, she can withdraw the full 100 Token A in the contract, and her deposit tokens are burned.

After this withdrawal:

- Token A balance in the protocol = 0
- Deposit tokens for Token A = 0
- Loan tokens for Token A = 60

This creates a broken state where the lending ratio becomes `60 / 0` = infinite. During the `assetsValueUpdate` function call, the lending ratio is forcibly set to `0` because of:

```
```

```
if (tempTotalSupply[0] > 0) {
    lendingRatio = tempTotalSupply[1] * 10000 / tempTotalSupply[0];
} else {
    lendingRatio = 0;
}
```

As a result, the interest rate also becomes `0`. This allows borrowers of Token A to avoid paying any interest.

The root cause is that the protocol relies directly on `balanceOf` for tracking token balances, instead of maintaining an internal accounting system.

Recommendation

Track balances internally instead of relying on raw `balanceOf` values.

Title: M-1 Inconsistency In Utilization Rate Enforcement Between Borrowing And Liquidation

Description

During liquidation, if the deposited assets of the account being liquidated have already been borrowed by other users, there may not be enough available balance in the protocol to complete the liquidation.

For example, assume Alice deposits 100 Token A and there are some Token B in the protocol. For simplicity, assume both tokens have the same price. Alice then borrows 80 Token B, and another user borrows 60 Token A (which was deposited by Alice). Later, due to a price increase in Token B, Alice becomes eligible for liquidation. The liquidator is expected to repay 80 Token B (Alice's debt) and receive 100 Token A (Alice's deposit). However, since 60 of the 100 Token A have already been borrowed by another user, there is insufficient Token A available in the protocol to fully liquidate Alice.

If the liquidator instead partially liquidates Alice, for example by only liquidating 40 Token A, then the utilization rate of Token A becomes 100% because there are only 60 Deposit Token and 60 Loan Token corresponding to token A. However, during the function `lendAsset`, the protocol enforces a maximum 99% utilization rate:

```
require(
    IERC20(assetsDepositAndLend[tokenAddr][0]).totalSupply() * 99 / 100 >=
    IERC20(assetsDepositAndLend[tokenAddr][1]).totalSupply(),
    "Lending Manager: total amount borrowed can t exceeds 99% of the deposit"
);
```

This creates an inconsistency: the utilization cap is enforced during borrowing but can be violated during liquidation.

Currently, the functions `tokenLiquidate` and `tokenLiquidateEstimate` do not account for the available balance in the protocol.

Recommendation

It is recommended to modify these functions to consider the actual available balance before performing liquidation.

Title: L-1 Inconsistent Asset Reference In Risk Isolation Mode Lending Validation

Description

When the function `lendAsset` is called, if a user's mode is set to `1`, it requires that the `tokenAddr` matches `riskIsolationModeAcceptAssets`:

```
if (userMode[user] == 1) {
    require(
        tokenAddr == riskIsolationModeAcceptAssets,
        "Lending Manager: Wrong Token in Risk Isolation Mode"
    );
    uint tempAmount =
        IERC20(assetsDepositAndLend[riskIsolationModeAcceptAssets][1])
            .balanceOf(user) + amountNormalize;

    riskIsolationModeLendingNetAmount[tokenAddr] =
        riskIsolationModeLendingNetAmount[tokenAddr]
        - userRIMAssetsLendingNetAmount[user][tokenAddr]
        + tempAmount;

    userRIMAssetsLendingNetAmount[user][tokenAddr] = tempAmount;

    require(
        riskIsolationModeLendingNetAmount[tokenAddr] <=
            licensedAssets[userRIMAssetsAddress[user]].maxLendingAmountInRIM,
        "Lending Manager: Deposit Amount exceed limits"
    );
}
```

At the end, the function checks that the total borrowed amount of the token does not exceed the maximum allowed limit:

```
require(
    riskIsolationModeLendingNetAmount[tokenAddr] <=
        licensedAssets[userRIMAssetsAddress[user]].maxLendingAmountInRIM,
    "Lending Manager: Deposit Amount exceed limits"
);
```

The problem is that the comparison references `userRIMAssetsAddress[user]` instead of `tokenAddr` (which is equal to `riskIsolationModeAcceptAssets`). Since `userRIMAssetsAddress[user]` can be freely set through `userModeSetting`, it may not align with `tokenAddr`:

```
function userModeSetting(
    uint8 _mode,
    address _userRIMAssetsAddress,
    address user
) public onlyInterface(user) {
    require(
        _userTotalLendingValue(user) == 0 &&
            _userTotalDepositValue(user) == 0,
        "Lending Manager: should return all Lending Assets and withdraw all
Deposit Assets."
    );

    if (_mode == 1) {
        require(
            licensedAssets[_userRIMAssetsAddress].maxLendingAmountInRIM > 0,
            "Lending Manager: Mode 1 Need a RIMAsset."
        );
    }

    userMode[user] = _mode;
    userRIMAssetsAddress[user] = _userRIMAssetsAddress;

    emit UserModeSetting(user, _mode, _userRIMAssetsAddress);
}
```

Recommendation

Either enforce that `_userRIMAssetsAddress` must equal `riskIsolationModeAcceptAssets` in `userModeSetting`, or replace `licensedAssets[userRIMAssetsAddress[user]].maxLendingAmountInRIM` with `licensedAssets[tokenAddr].maxLendingAmountInRIM` in `lendAsset()`.

Title: L-2 Risk Isolation Lending Limit Can Be Exceeded Due To Accrued Interest

Description

When the function `lendAsset` is invoked for a user whose mode is 1, it enforces that `riskIsolationModeLendingNetAmount[tokenAddr]` does not exceed `licensedAssets[userRIMAssetsAddress[user]].maxLendingAmountInRIM`:

```
        if(userMode[user] == 1){
            require(tokenAddr == riskIsolationModeAcceptAssets,"Lending Manager:
```

```
Wrong Token in Risk Isolation Mode");
            uint tempAmount =
IERC20(assetsDepositAndLend[riskIsolationModeAcceptAssets][1]).balanceOf(user) +
amountNormalize;
            riskIsolationModeLendingNetAmount[tokenAddr] =
riskIsolationModeLendingNetAmount[tokenAddr]
                                                        -
userRIMAssetsLendingNetAmount[user][tokenAddr]
                                                        + tempAmount;
            userRIMAssetsLendingNetAmount[user][tokenAddr] = tempAmount;
            require(riskIsolationModeLendingNetAmount[tokenAddr] <=
licensedAssets[userRIMAssetsAddress[user]].maxLendingAmountInRIM,"Lending Manager:
Deposit Amount exceed limits");
        }
```

However, because interest accrues on borrowed amounts, `riskIsolationModeLendingNetAmount[tokenAddr]` can later exceed `licensedAssets[userRIMAssetsAddress[user]].maxLendingAmountInRIM`. For instance, a user might borrow exactly up to the maximum allowed amount, and as interest accrues, the net lending amount surpasses the limit. This situation can occur even if the user is not in liquidation.

The effect is not immediate, but it violates the intended safety constraint that `riskIsolationModeLendingNetAmount[tokenAddr]` must remain below the maximum limit.

Recommendation

It is recommended to implement measures to ensure this limit is not breached due to accrued interest.

Title: I-1 Unvalidated Oracle Responses May Lead To Incorrect Lending Value Calculations

Description

The oracle being used in the protocol is not explicitly defined, which creates ambiguity when its values are relied upon, especially during sensitive operations such as liquidations or loan borrowing. In the function `_userTotalLendingValue`, the protocol directly queries the oracle without handling potential failures or unexpected behavior:

```
function _userTotalLendingValue(address _user) internal view returns(uint values){
    //require(assetsSerialNumber.length < 100,"Lending Manager: Too Much assets");
    for(uint i=0;i<assetsSerialNumber.length;i++){
        values += IERC20(assetsDepositAndLend[assetsSerialNumber[i]]
[1]).balanceOf(_user)
        * iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether;
    }
}
```

If the oracle returns invalid, stale, or erroneous data, it could lead to incorrect lending value calculations and, in turn, faulty liquidations or borrowing decisions.

Recommendation

It is recommended to implement proper validation and error handling for oracle responses to ensure protocol safety.

Title: I-2 Interface Removal Does Not Break Loop If Found Or Revert If Not Found

Description

When removing an interface, if the target interface is found, the function replaces it with the last element in the `interfaceArray` and pops the array:

```
function xInterfacesetting(address _xInterface, bool _ToF) external onlySetter {
    require(isContract(_xInterface), "Lending Manager: Interface MUST be a
contract.");
    uint lengthTemp = interfaceArray.length;
    if (_ToF == false) {
        for (uint i = 0; i != lengthTemp; i++) {
            if (interfaceArray[i] == _xInterface) {
                interfaceArray[i] = interfaceArray[lengthTemp - 1];
                interfaceArray.pop();
            }
        }
    } else if (xInterface[_xInterface] == false) {
        xInterface[_xInterface] = true;
        interfaceArray.push(_xInterface);
    }
    emit InterfaceSetup(_xInterface, _ToF);
}
```

The issue is that after finding and removing the interface, the loop continues unnecessarily. Additionally, if the interface is not found, the function does not revert to inform the caller that the removal was unsuccessful.

Recommendation

It is recommended to break the loop once the interface is found and removed, and revert the transaction if the interface was not found.

Title: I-3 Confusing Error Message In `lendAsset` To Reflect Borrow Limit Exceeded

Description

The error message in the `lendAsset` function is misleading:

```
require(riskIsolationModeLendingNetAmount[tokenAddr] <=
licensedAssets[userRIMAssetsAddress[user]].maxLendingAmountInRIM,"Lending Manager:
Deposit Amount exceed limits");
```

Recommendation

It is recommended to update the message to accurately indicate that the borrow amount exceeds the allowed limit.

Title: I-4 Misuse Of “Lend” Instead Of “Borrow” In Protocol Logic

Description

In the entire protocol, the term **"lend"** is used, which conventionally means **"to provide an asset to someone else (or a protocol) with the expectation of getting it back, usually with interest."** However, the actual implementation of the logic corresponds to **"borrow,"** which means **"to take an asset (such as money or a token) from someone else with the obligation to return it later, typically with interest."**

Recommendation

It is recommended to replace all instances of **"lend"** with **"borrow"** throughout the codebase to accurately reflect the intended functionality.

Title: I-5 `_userRIMAssetsAddress` Not Reset When Leaving Mode 1

Description

When a user switches their mode from `1` to any other value, it would be safer to reset `_userRIMAssetsAddress` to `address(0)`. This change has no effect on deposit or borrow behavior, since `_userRIMAssetsAddress` is only relevant when the user’s mode is `1`, but it keeps the code more clear.

```
function userModeSetting(
    uint8 _mode,
    address _userRIMAssetsAddress,
    address user
) public onlyInterface(user) {
    require(
        _userTotalLendingValue(user) == 0 && _userTotalDepositValue(user) == 0,
        "Lending Manager: should return all Lending Assets and withdraw all
Deposit Assets."
```

```
        );

        if (_mode == 1) {
            require(
                licensedAssets[_userRIMAssetsAddress].maxLendingAmountInRIM > 0,
                "Lending Manager: Mode 1 Need a RIMAsset."
            );
        }

        userMode[user] = _mode;
        userRIMAssetsAddress[user] = _userRIMAssetsAddress;
        emit UserModeSetting(user, _mode, _userRIMAssetsAddress);
    }
```

Recommendation

Reset `_userRIMAssetsAddress` to zero when the mode is changed from `1` to any other value.